PŘÍRODOVĚDECKÁ FAKULTA UNIVERZITY PALACKÉHO
KATEDRA INFORMATIKY

DIPLOMOVÁ PRÁCE

Pokročilé metody fotorealistického zobrazování

2013 Ondřej Bahounek

Anotace

Mezi pokročilé metody fotorealistického zobrazování patří různé techniky optimalizace ray-tracingu. Nejefektivnější a algoritmicky nejzajímavější z nich jsou především techniky založené na principu hierarchie akcelerujících struktur, které má tato práce za cíl prozkoumat a porovnat. Pozornost je věnována odlišnostem metod dělících prostor (např. kd-tree) od metod dělících objekty (použitím např. R-tree). Prozkoumány jsou však i optimalizace výpočtů průsečíků paprsku s objekty nebo optimalizace vržených stínů. Pro prezentaci výsledků a konstrukci scén byl vytvořen renderovací program s trojrozměrným editorem.

Děkuji vedoucímu diplomové práce Mgr. Eduardu Bartlovi, Ph.D. za cenné rady a trpělivost při spolupráci, jež mi velmi pomohla při tvorbě teoretických i praktických částí práce.

Obsah

1. Úvod **8**

2. Základní pojmy a principy **9**
2.1. Transformace . . . 9
2.1.1. Homogenní souřadnice . . . 10
2.1.2. Afinní transformace . . . 10
2.2. Eulerovy úhly . . . 13
2.3. Kvaterniony . . . 14
2.4. Průsečík paprsku s objektem . . . 17
2.4.1. Osově orientovaný kvádr (AABB) . . . 18
2.4.2. Obecně orientovaný kvádr (OBB) . . . 18
2.4.3. Kužel . . . 21
2.4.4. Obecný objekt . . . 25
2.4.5. Trojúhelníková ploška . . . 25

3. Optimalizace ray-tracingu **33**
3.1. Ray-tracing . . . 33
3.2. Optimalizační principy . . . 36
3.2.1. Porovnávání optimalizačních metod . . . 36
3.3. Optimalizace kódu . . . 38
3.4. Akcelerující struktury . . . 39
3.4.1. Obálka (BV) . . . 40
3.4.2. Hierarchie obálek (BVH) . . . 43
3.4.3. R-tree . . . 45
3.4.4. Mřížka . . . 51
3.4.5. Octree . . . 53
3.4.6. Kd-tree . . . 54
3.4.7. BSP-tree . . . 56
3.4.8. Srovnání metod . . . 56
3.5. Optimalizace vržených stínů . . . 56

4. Editor **60**
4.1. Pohled kamery . . . 60
4.2. Body elipsy . . . 61

Závěr **63**

Conclusions **64**

Reference **65**

A. Program RayTracer **66**
A.1. Ovládání editoru . . . 66
A.2. Import a export dat . . . 67
A.3. Animace . . . 67

B. Obsah přiloženého DVD **68**

Seznam obrázků

1. Osově orientovaný kvádr – AABB 18
2. Transformace normály 21
3. Průsečík paprsku s kuželem 24
4. Obecný objekt 25
5. Normály plošek objektu 26
6. Ukázka interpolace normál 26
7. Způsoby výpočtů normál 27
8. Interpolace normál 30
9. Model Australské Královské Koruny 33
10. Porovnání naivního principu s optimalizací 38
11. Typy obálek 1 41
12. Typy obálek 2 41
13. Hraniční roviny 43
14. Obrázek k demonstraci neefektivity samostatných obálek 45
15. BVH 46
16. Mřížka 51
17. Nevýhoda mřížky 52
18. Tvorba minimální bounding volume pro octree 55
19. Scéna k porovnání metod 1 57
20. Scéna k porovnání metod 2 57
21. Pohled kamery 60
22. Vykreslení elipsy v editoru 62
23. Statistika a struktura kódu 66

Seznam tabulek

1. Porovnání naivního principu s optimalizací k obrázku 10. 37
2. Nevýhoda samostatných obálek – k obrázku 14.a. 44
3. Nevýhoda samostatných obálek – k obrázku 14.b. 44
4. Data ke scéně z obrázku 19. 58
5. Data ke scéně z obrázku 20. 58
6. Porovnání optimalizací vržených stínů 59
7. Ovládání editoru 67

1. Úvod

Fotorealistické zobrazování má být dle své definice nerozlišitelné od reálně pořízených snímků. Pozorovatel by neměl poznat, že se dívá na uměle vytvořený obraz, anebo při pozorování neexistujícího prostředí by měl uvěřit v jeho možnou existenci. Což je důležité zejména při filmových vizuálních efektech, kdy se mísí hraná a počítačová tvorba do jednoho obrazu. Existuje několik prostředků, jak dosáhnout reálně vypadajícího efektu, vyvíjejících se doslova každý den díky novým technologickým možnostem.

V poslední době je kladen důraz hlavně na výkonnost fotorealistického zobrazování. Především ve filmovém průmyslu je častěji požadováno vidět komponovaný obraz v reálném čase – ještě v průběhů natáčení, nikoli až ve fázi postprocesingu. Původní naivní technika ray tracingu je takřka neschopna utvářet obrázky složitějších scén v reálném čase i při neustálém vývoji počítacích strojů. Předmětem zájmu této práce je prozkoumat a porovnat populární optimalizační techniky fotorealistického zobrazování navázáním na předchozí bakalářskou práci [2]. Proto zde již nebudou uvedeny základy algoritmů sledování paprsku ani Phongova osvětlovacího modelu. Je velmi doporučeno se s předchozí prací, nebo alespoň s těmito tématy, seznámit před dalším čtením. Průvodním softwarem bakalářské byl renderovací program RayTracer, jehož jádro bylo nyní za cíl podstoupit zde zkoumaným optimalizačním principům.

Text je opět rozdělen do tří hlavních částí. Úvodní kapitola (2.) pojednává o matematických principech používaných v ray-tracingu, především větší část je věnována metodám výpočtu průsečíku paprsku s objekty scény. Kromě některých připomenutí předchozí práce obsahuje i pokročilé matematické struktury jako homogenní matice nebo kvaterniony.

Následující kapitola (3.) se zabývá hlavním tématem práce, a tedy optimalizačními technikami. Způsobů, jak optimalizaci začlenit do původních naivních algoritmů ray-tracingu, je však celá řada. Nejprve je představíme obecně a popíše smysl jejich porovnávání. Metody na porovnávání efektivity optimalizací však nejsou tak zřejmé, jak by se na první pohled zdálo. Následně vybereme optimalizační metody založené na hierarchii obálek (BVH), které detailněji prozkoumáme a mezi sebou porovnáme.

Předmětem zájmu pokročilých grafických metod je i tvorba interaktivního trojrozměrného editoru umožňujícího uživatelsky přívětivě vytvářet scény pro ray-tracing a specifikovat jeho výstupy. Kapitola 4. je věnována tvorbě editoru a jeho očekávaných funkcionalit. Nakonec (v kapitole A.) jsou předvedeny některé vlastnosti programu, s jehož výstupy se setkáváme téměř v každé části práce. Nový program, vzniklý sloučením ray-tracingového jádra s editorem, se nazývá RayTracer verze 2 a je součástí příloh práce na DVD společně s ukázkami výstupů, které jsou v podobě obrázků i animací.

2. Základní pojmy a principy

K pochopení hlavních principů algoritmu ray-tracingu je potřeba porozumět nemalému množství definic a výpočtů v analytické geometrii, jejichž znalost je již při čtení práce předpokládána. Některé zde zopakujeme, ale je doporučeno prostudování i předešlé práce [2].

2.1. Transformace

Geometrické transformace mají pro počítačovou grafiku značný význam. Jejich aplikací na souřadnice bodů objektu jej můžeme různě měnit. Mezi *lineární transformace* řadíme například

- změna měřítka (scale),
- zkosení (skew/shear),
- rotace.

Nelineární jsou například

- posunutí (translace),
- perspektivní projekce.

Tedy pro nás významné pozorování je, že posunutí nepatří mezi lineární transformace. Aplikací transformace na bod P s kartézskými souřadnicemi $[x, y, z]$ získáme bod P' o souřadnicích $[x', y', z']$. Transformace objektu znamená aplikaci operace transformace na všechny jeho body nebo i na parametry, nimiž je popsán.

Víme, že lineární zobrazení F musí splňovat:

$$\begin{aligned} F(\vec{a} + \vec{b}) &= F(\vec{a}) + F(\vec{b}), \\ F(k \cdot \vec{a}) &= k \cdot F(\vec{a}), \end{aligned}$$

kde $\vec{a}, \vec{b} \in \mathbb{R}^n$, $k \in \mathbb{R}$. Budeme-li zobrazení reprezentovat maticí $\mathbf{M}$ o rozměru 3×3, můžeme definovat lineární transformace. Nelineární transformace v 3D prostoru nemohou být implementovány použitím matic o rozměru 3×3. Množina transformací, která obsahuje jak lineární transformace, tak nelineární (jako je posunutí), se nazývá *afinní* transformace. Jedná se o nejobecnější třídu transformací, kterou budeme zvažovat, a tato transformace má tvar:

$$P' = \mathbf{M} \cdot P + \vec{b}.$$

$\mathbf{M}$ nazveme transformační maticí (např. matice rotace) a $\vec{b} \in \mathbb{R}^n$ je vektor posunutí. Transformace F je invertibilní, existuje-li k ní transformace, která vrací změny provedené původní transformací, tzn.:

$$F^{-1}(F(\vec{a})) = \vec{a},$$

pro každé $\vec{a} \in \mathbb{R}^n$. U lineárních transformací stačí tedy najít inverzní matici, ale např. u rotační matice $\mathbf{R}$ navíc platí, že je ortogonální, a tedy její inverzní maticí je matice transponovaná:

$$\mathbf{R}^{-1} = \mathbf{R}^T.$$

Nelineární transformace nemá jednotný způsob inverzní transformace (některé dokonce nejsou invertibilní – např. perspektivní projekce), ale nám bude stačit umět najít pouze inverzi k nelineární transformaci posunutí, což se provede jednoduše posunutím o opačný směr, než původní posunutí.

2.1.1. Homogenní souřadnice

Počítačová grafika potřebuje efektivně a jednotně popsat lineární i nelineární transformace. Zavedení homogenních souřadnic pro reprezentace bodů místo kartézských souřadnic práci se všemi transformacemi značně zjednodušuje. Homogenní souřadnice tvoří základ projektivní geometrie, použité převážně při projekci trojrozměrných scén do dvourozměrné roviny. Vzhledem k tomu, že tato práce nemá za cíl poskytnout úplné základy lineární algebry, značnou část teorie vynecháme. Zaměříme se navíc jen na transformace v 3D prostoru.

Hlavní rozdíl oproti kartézských souřadnic spočívá v tom, že $n + 1$ souřadnic bude reprezentovat n-rozměrný bod. Konkrétně v 3D bod nyní nemá souřadnice 3, ale 4:

$$P = [x, y, z, w].$$

První tři souřadnice jsou opět kartézské, čtvrtá vyjadřuje tzv. *váhu*. Je-li váha bodu nenulová, mluvíme o tzv. *vlastním* bodu. Jinak se jedná o bod *nevlastní*. Bodu se obyčejně přiřazuje váha 1 a rozdíl dvou bodů (udávající vektor) má váhu 0. Základní operací pro homogenní souřadnice $[x', y', z', w]$ je jejich převod do kartézských souřadnic $[x, y, z]$:

$$\begin{aligned} x &= \frac{x'}{w}, \\ y &= \frac{y'}{w}, \\ z &= \frac{z'}{w}. \end{aligned}$$

2.1.2. Afinní transformace

Víme, že afinními transformacemi homogenních souřadnic můžeme provádět jak lineární transformace, tak i některé nelineární transformace (např. posunutí). To ale není jediná výhoda. Důležitou vlastností z implementačního hlediska je možnost transformace skládat do jedné matice a výslednou složenou transformaci aplikovat na všechny body objektu pouze jednou. Skládání transformací je realizováno násobením jejich transformačních matic, přičemž záleží na jejich pořadí.

Násobíme-li matici souřadnicemi zprava, skládáme matice jejich přidáváním zleva. Například, chceme-li provést rotaci objektu (daného bodem P) kolem své osy maticí $\mathbf{R}$, přesunout jej do světových souřadnic (maticí $\mathbf{T}_w$) a nakonec celý vnější svět ještě přiblížit maticí $\mathbf{S}$, násobení matic bude v tomto pořadí:

$$P' = \mathbf{S} \cdot \mathbf{T}_w \cdot \mathbf{R} \cdot P.$$

Pokud chceme objekt rotovat kolem jiného bodu, maticí $\mathbf{T}_l$ jej tam přesuneme a po rotaci vrátíme zpět na původní pozici (maticí $\mathbf{T}_l^{-1}$). Výsledná posloupnost matic bude následující:

$$P' = \mathbf{S} \cdot \mathbf{T}_w \cdot \mathbf{T}_l^{-1} \cdot \mathbf{R} \cdot \mathbf{T}_l \cdot P.$$

Nelokální transformace můžeme vyjádřit jedinou transformační maticí a aplikovat ji na všechny objekty světa. Stejně tak lze vyjádřit jedinou lokální maticí každého objektu jeho lokální podobu. Složením obou matic můžeme jedinou transformací upravit každý bod pouze jediným násobením s maticí. Je zřejmé, že výpočetní přínos je značný, obzvlášť, máme-li miliony bodů všech objektů.

Dále uvedeme podoby transformačních matic k jednotlivým transformacím. Posunutí o Δx, Δy, Δz:

$$\begin{bmatrix} 1 & 0 & 0 & \Delta x \\ 0 & 1 & 0 & \Delta y \\ 0 & 0 & 1 & \Delta z \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

Změna měřítka:

$$\begin{bmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

Faktory škálování nemusí být všechny stejné a jejich význam je pro absolutní hodnotu v intervalu $(0, 1)$ zmenšení/zkrácení, jsou-li větší než 1, mají efekt zvětšení/prodloužení, a jsou-li záporné, způsobí zrcadlové zobrazení. Vynulováním právě jednoho z faktorů škálování získáme tzv. ortografickou projekci na projekční rovinu tvořenou nenulovými faktory. Zkosení podél osy x:

$$\begin{bmatrix} 1 & k_{xy} & k_{kz} & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix},$$

zkosení podél osy y:

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ k_{yx} & 1 & k_{yz} & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix},$$

zkosení podél osy z:

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ k_{zx} & k_{zy} & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

Rotace okolo osy x o úhel ψ:

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\psi & \sin\psi & 0 \\ 0 & -\sin\psi & \cos\psi & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix},$$

rotace okolo osy y o úhel θ:

$$\begin{bmatrix} \cos\theta & 0 & -\sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ \sin\theta & 0 & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix},$$

rotace okolo osy z o úhel ϕ:

$$\begin{bmatrix} \cos\phi & \sin\phi & 0 & 0 \\ -\sin\phi & \cos\phi & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

Perspektivní promítání s průmětnou procházející počátkem a kolmou na osu z a pozorovatelem umístěném v bodě z_0:

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & \frac{1}{z_0} & 1 \end{bmatrix}.$$

Chceme-li promítání přibližovat a oddalovat, můžeme rovnou přidat i škálování:

$$\begin{bmatrix} scale & 0 & 0 & 0 \\ 0 & scale & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & \frac{1}{z_0} & 1 \end{bmatrix}.$$

2.2. Eulerovy úhly

V kapitole 2.1.2. jsme si představili tři různé rotační matice s rotačními úhly ψ, θ a ϕ. Tyto úhly se nazývají *Eulerovy úhly*. Každou rotační matici je možné vyjádřit jako složení rotačních matic kolem osy x, y a z. Poskládáním rotačních matic můžeme určit, jestli chceme levotočivé nebo pravotočivé osy souřadnicového systému. Nejčastěji se volí pořadí $x - y - z$. Problém je, že Eulerovy úhly nejsou dány jednoznačně. To znamená, že k dané orientaci existuje více trojic Eulerových úhlů a tedy i způsobů, jak bod rotovat na určitou pozici. V počítačové grafice je často potřeba umět vyjádřit z dané rotační matice jednoznačnou trojici Eulerových úhlů. Tato technika se nazývá *faktorizace matice* a je velmi důležitá zejména při tvorbě grafických editorů. Abychom zaručili jednoznačnost úhlů, převedeme je vždy do kanonické formy, kde $\psi \in [-180, 180]$, $\theta \in [-90, 90]$ a $\phi \in [-180, 180]$. Pro jakoukoli orientaci existuje vždy jedna kanonická trojice Eulerových úhlů. Představíme jedno z možných řešeních podle principu uvedeného v [11].

Nechť byla rotační matice $\mathbf{R}$ složena rotacemi okolo os x (o úhel ψ), y (o úhel θ) a z (o úhel ϕ):

$$\mathbf{R} = \begin{bmatrix} R_{11} & R_{12} & R_{13} \\ R_{21} & R_{22} & R_{23} \\ R_{31} & R_{32} & R_{33} \end{bmatrix}.$$

Pak vyjádřením každého prvku matice $\mathbf{R}$ odpovídajícím výrazem obdržíme 9 rovnic o 3 neznámých:

$$\begin{aligned}
R_{11} &= \cos\theta \cdot \cos\phi, \\
R_{12} &= \sin\psi \cdot \sin\theta \cdot \cos\phi - \cos\psi \cdot \sin\phi, \\
R_{13} &= \cos\psi \cdot \sin\theta \cdot \cos\phi + \sin\psi \cdot \sin\phi, \\
R_{21} &= \cos\theta \cdot \sin\phi, \\
R_{22} &= \sin\psi \cdot \sin\theta \cdot \sin\phi + \cos\psi \cdot \cos\phi, \\
R_{23} &= \cos\psi \cdot \sin\theta \cdot \sin\phi - \sin\psi \cdot \cos\phi, \\
R_{31} &= -\sin\theta, \\
R_{32} &= \sin\psi \cdot \cos\theta, \\
R_{33} &= \cos\psi \cdot \cos\theta.
\end{aligned}$$

Nejdříve můžeme rovnou vypočítat úhel θ:

$$\begin{aligned}
R_{31} &= -\sin\theta, \\
\theta_1 &= -\sin^{-1} R_{31}, \\
\theta_2 &= \pi + \sin^{-1} R_{31}.
\end{aligned}$$

Další úhly se získají pomocí úhlu θ. Musí se ale ošetřit případ, kdy výraz R_{31} je roven 1 nebo -1. Úplný algoritmus viz 2.1.. Používáme v něm programátorskou

funkci $\arctan 2(x, y)$ sloužící např. k získání informace o znaménku ze vstupních hodnot.

Algoritmus 2.1. Výpočet Eulerových úhlů z rotační matice

Vstup: Rotační matice R
Výstup: Eulerovy úhly ψ, θ, ϕ

if $R_{31} \neq \pm 1$ **then**
  $\theta_1 = -\arcsin(R_{31})$
  $\theta_2 = \pi - \theta_1$
  $\psi_1 = \arctan 2(\frac{R_{32}}{\cos\theta_1}, \frac{R_{33}}{\cos\theta_1})$
  $\psi_2 = \arctan 2(\frac{R_{32}}{\cos\theta_2}, \frac{R_{33}}{\cos\theta_2})$
  $\phi_1 = \arctan 2(\frac{R_{21}}{\cos\theta_1}, \frac{R_{11}}{\cos\theta_1})$
  $\phi_2 = \arctan 2(\frac{R_{21}}{\cos\theta_2}, \frac{R_{11}}{\cos\theta_2})$
else
  $\phi = 0$
  if $R_{31} = -1$ **then**
    $\theta = \frac{\pi}{2}$
    $\psi = \phi + \arctan 2(R_{12}, R_{13})$
  else
    $\theta = -\frac{\pi}{2}$
    $\psi = -\phi + \arctan 2(-R_{12}, -R_{13})$
  end if
end if

Takovéto řešení je odolné proti jevu nazvanému *Gimbal lock*, jenž se v našem případě vyskytne, když $\theta = \pm\frac{\pi}{2}$. Tento nepříjemný jev je dobře známý a nastane, když se z prostřední rotační matice stane jednotková. Prakticky to znamená, že při rotaci kolem jedné osy se provede rotace i okolo druhé o stejný úhel. Má to souvislost s nejednoznačností Eulerových úhlů, a i když budeme pracovat jen s kanonickými úhly, tento případ musíme zvlášť ošetřit.

2.3. Kvaterniony

V předchozí kapitole jsme popsali jeden z možných způsobů, jak reprezentovat orientaci v 3D prostoru – rotační matici. Tuto orientaci lze faktorizovat na pouhá tři čísla – Eulerovy úhly – přičemž se jedná o nejmenší možný počet čísel, jimiž lze reprezentovat orientaci v 3D. Spjaté problémy typu Gimbal lock vedou k potřebě jednoznačné reprezentace. Řešením je zvýšit počet parametrů popisujících orientaci. Dostáváme se tak k pojmu *kvaternion*. Kvaternion q má dvě části: skalární (w) a vektorovou ($\vec{v} = [x, y, z]$). Celkem jej tedy tvoří 4 složky

a jeho možné zápisy jsou:

$$\begin{aligned} q &= (w, x, y, z), \\ q &= (w, \vec{v}). \end{aligned}$$

Na kvaternion lze také pohlížet jako na komplexní číslo se třemi imaginárními složkami i, j, k s vlastnostmi:

$$\begin{gathered} i^2 = j^2 = k^2 = -1, \\ ij = k, ji = -k, \\ jk = i, kj = -i, \\ ki = j, ik = -j. \end{gathered}$$

Pak kvaternion $q = (w, x, y, z)$ definuje komplexní číslo $w + xi + yj + zk$.

Mějme dva kvaterniony $q_1 = (w_1, x_1, y_1, z_1)$ a $q_2 = (w_2, x_2, y_2, z_2)$. Operace **součet** a **rozdíl** se provede jako

$$q_1 \pm q_2 = (w_1 \pm w_2) + (x_1 \pm x_2)i + (y_1 \pm y_2)j + (z_1 \pm z_2)k$$

a jejich **součin**

$$\begin{aligned} q_1 \times q_2 &= (w_1 w_2 - \vec{v_1} * \vec{v_2}, w_2 \vec{v_1} + w_1 \vec{v_2} + \vec{v_1} \times \vec{v_2}) \\ &= (w_1 w_2 - x_1 x_2 - y_1 y_2 - z_1 z_2) + (w_1 x_2 + x_1 w_2 + y_1 z_2 - z_1 y_2)i + \\ &\quad + (w_1 y_2 - x_1 z_2 + y_1 w_2 + z_1 x_2)j + (w_1 z_2 + x_1 y_2 - y_1 x_2 + z_1 w_2)k, \end{aligned}$$

přičemž násobení není komutativní: $q_1 \times q_2 \neq q_2 \times q_1$. Operace $*$ značí **skalární součin** vektorů. **Norma** kvaternionu se vypočítá jako

$$\|q\|^2 = w^2 + x^2 + y^2 + z^2,$$

což je reálné číslo stejně jako **skalární součin** kvaternionů q_1 a q_2:

$$q_1 * q_2 = w_1 w_2 + x_1 x_2 + y_1 y_2 + z_1 z_2.$$

Velikost kvaternionu je dána odmocninou normy:

$$\|q\| = \sqrt{w^2 + x^2 + y^2 + z^2}.$$

Kvaternion je jednotkový, je-li jeho velikost rovna 1. Nakonec, kvaternion velikosti 1 lze též vyjádřit jako

$$q = (\cos\frac{\theta}{2}, \sin\frac{\theta}{2} \cdot \vec{n}) = (\cos\frac{\theta}{2}, \sin\frac{\theta}{2} \cdot n_x, \sin\frac{\theta}{2} \cdot n_y, \sin\frac{\theta}{2} \cdot n_z). \quad (1)$$

Rovnice (1) vyžaduje vysvětlení. Již Euler dokázal, že posloupnost rotací je ekvivalentní jediné rotaci okolo libovolné osy. Touto osou může být libovolný rotační

vektor – zde označený jako $\vec{n}$, jehož délku zvažujme 1 – a úhel, o který se okolo této osy otáčíme, je právě θ. Je důležité si uvědomit, že skalární komponenty w a θ jsou mezi sebou úzce spjaté, ale nejsou identické. To samé platí i pro vektory $\vec{v}$ a $\vec{n}$. Tedy ještě jednou pro zdůraznění:

$$\begin{aligned} w &= \cos\frac{\theta}{2}, \\ v_x &= \sin\frac{\theta}{2}\cdot n_x, \\ v_y &= \sin\frac{\theta}{2}\cdot n_y, \\ v_z &= \sin\frac{\theta}{2}\cdot n_z. \end{aligned}$$

Je jednoduché ověřit, že takto definovaný kvaternion je vždy jednotkový:

$$\begin{aligned} \|q\| &= \sqrt{w^2 + \|\vec{v}\|^2} \\ &= \sqrt{\cos^2\frac{\theta}{2} + \sin^2\frac{\theta}{2}\cdot\|\vec{n}\|^2} \\ &= \sqrt{\cos^2\frac{\theta}{2} + \sin^2\frac{\theta}{2}\cdot 1} \\ &= \sqrt{1} \\ &= 1. \end{aligned}$$

Je-li kvaternion jednotkový, tak jeho **inverze** se spočítá jako

$$\begin{aligned} (w,\vec{v})^{-1} &= (\cos\frac{\theta}{2}, \sin\frac{\theta}{2}\cdot\vec{n}) \\ &= (\cos\left(-\frac{\theta}{2}\right), \sin\left(-\frac{\theta}{2}\right)\cdot\vec{n}) \\ &= (\cos\frac{\theta}{2}, -\sin\frac{\theta}{2}\cdot\vec{n}) \\ &= (w, -\vec{v}). \end{aligned}$$

O kvaternionu řekneme, že je **jednotkou**, je-li ve tvaru $(1,0,0,0)$. Pro součin kvaternionu a jeho inverze pak platí, že

$$q \times q^{-1} = (1,0,0,0).$$

Rotační matici R lze získat z kvaternionu jako

$$R = \begin{bmatrix} 1-2y^2-2z^2 & 2xy+2wz & 2xz-2wy \\ 2xy-2wz & 1-2x^2-2z^2 & 2yz+2wx \\ 2xz+2wy & 2yz-2wx & 1-2x^2-2y^2 \end{bmatrix}. \tag{2}$$

Jednoznačné **Eulerovy úhly** získáme jako

$$\begin{aligned}\psi &= \arctan 2(2(wx+yz), 1-2(x^2+y^2)),\\ \theta &= \arcsin(2(wy-zx)),\\ \phi &= \arctan 2(2(zx+xy), 1-2(y^2+z^2)).\end{aligned}$$

Právě jsme popsali základní teorii o kvaternionech, častým jejich využitím v počítačové grafice je ovšem možnost převádět jeden vektor na druhý. Přesněji, ze dvou vektorů vytvořit kvaternion, jenž v rovině dané těmito vektory rotuje všechny body o úhel, který tyto vektory svírají. Nebo ještě jinak: máme vektor $\vec{v_1}$ směřující jedním směrem a chceme vytvořit kvaternion, který jej bude rotovat na jiný směr $\vec{v_2}$: Nechť jsou $\vec{v_1}$ i $\vec{v_2}$ normalizovány. Vytvoříme k nim kolmý (rotační) vektor $\vec{r} = (r_x, r_y, r_z)$ a následně kvaternion $q = (w, x, y, z)$:

$$\begin{aligned}s &= \sqrt{2(1+\vec{v_1} * \vec{v_2})},\\ \vec{r} &= \frac{1}{s}\cdot(\vec{v_1}\times\vec{v_2}),\\ w &= \frac{s}{2},\\ x &= r_x,\\ y &= r_y,\\ z &= r_z.\end{aligned}$$

Jako θ by stačilo vzít pouze $\arccos(\vec{v_1} * \vec{v_2})$, ovšem při téměř rovnoběžných vektorech bychom měli s výpočtem problémy.

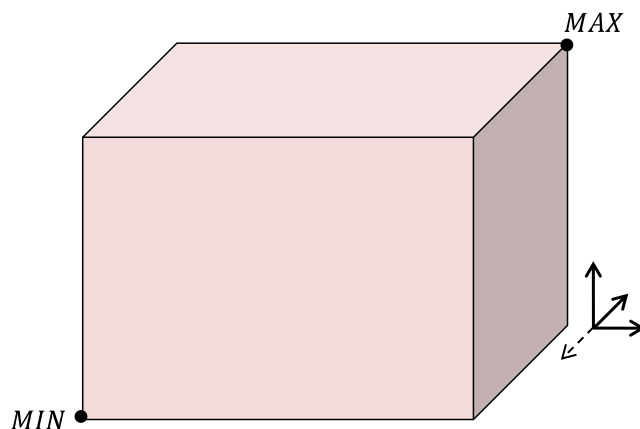
Občas je potřebné převést na kvaternion rotační matici, ale tato metoda i detailnější teorie o kvaternionech a jejich využití je popsána např. v [3].

2.4. Průsečík paprsku s objektem

V bakalářské práci jsme představili některé základní objekty popsané implicitními rovnicemi. Mezi tyto objekty patřily: koule, válec, krychle (rovnoosá i obecně orientovaná) a rovina. Často se využívá parametrického vyjádření objektů pro efektivnější výpočty v počítači. Ke všem objektům potřebujeme umět zjistit, zda jej protíná paprsek, v jakém bodě a jaký je v něm normálový vektor. Průsečíků je většinou více a vybírá se ten nejbližší k počátku paprsku v kladném směru. Všechny objekty měly zcela jasný a poměrně jednoduchý výpočet – až na obecně orientovanou krychli (dále jen obecná krychle nebo OBB). Ten probíhal tak, že se vypočítal průsečík paprsku s 6 rovinami krychle tvořících jeho stěny a následně ověřil, zda bod za průsečíkem leží uvnitř krychle. Je zřejmé, že algoritmus byl značně výpočetně neefektivní. Zato byl ale velmi intuitivní, což v této části není předností. Uvedeme tedy novou metodu výpočtu průsečíku paprsku s obecnou krychlí a obohatíme objekty o kužel a trojúhelník – z něhož následně uděláme libovolně deformovaný objekt.

2.4.1. Osově orientovaný kvádr (AABB)

Nejjednodušší trojrozměrný objekt, jenž může být protnut paprskem je rovnoosý kvádr (v počítačové grafice označována jako AABB – *axis aligned bounding box* – viz obrázek 1.). AABB má každou stěnu rovnoběžnou s některou souřadnicovou osou. Jedná se o jeden z nejvýznamnějších objektů vzhledem k jeho vlastnosti rychlého a snadného výpočtu průsečíku s paprskem, jenž obsahuje nejvýše 6 operací dělení. AABB stačí definovat pouze dvěma body: MIN a MAX udávající minimum a maximum pro každou souřadnici. Uvedeme zde téměř kompletní algoritmus (viz algoritmus 2.2.), aby bylo dobře vidět, jak moc je výpočetně jednoduchý. Samozřejmě, že jej lze ještě zefektivnit, ale to už by nebyl tak přehledný. Navíc častěji u tohoto objektu nám stačí pouze zjistit, zda jej paprsek protíná, takže výpočet normály se může vypustit. Menší problém ale spočívá v tzv. negativní nule, kde podle standardu IEEE 754 platí $-0 = 0$, což způsobí přehození intervalů a tedy nesprávné neprotnutí s paprskem. Řešení existuje (viz [12]). K tomuto objektu se vrátíme v kapitole 3. a vyjasníme i jeho významnost.



Obrázek 1.: **Osově orientovaný kvádr – AABB** (Axis aligned bounding box).

2.4.2. Obecně orientovaný kvádr (OBB)

Původní metoda výpočtu průniku paprsku s obecnou krychlí byla velmi neefektivní, což je teď patrné ve srovnání s jednoduchým AABB výpočtem. Vlastností, kdy objekt v základní poloze má mnohem jednodušší výpočet než v obecné poloze, disponují téměř všechny objekty (kromě koule). My jsme doposud u všech objektů uváděli obecný případ, jelikož jeho výpočet se od základního výpočetně moc nelišil a co do efektivity výpočtu nevyčníval. Co když ale máme objekt pouze v základní poloze (například AABB objekt) a chceme jej mít libovolně posunutý a orientovaný (např. OBB – *oriented bounding box*)? Umíme i k němu efektivně

Algoritmus 2.2. Průnik AABB s paprskem

Vstup: Paprsek($P0$, Pd), MIN, MAX
Výstup: průsečík P, normála N

```
for i = 0 to i < 3 do
  if Pd[i] ≥ 0 then
    t_min[i] = (MIN[i] − P0[i]) / Pd[i]
    t_max[i] = (MAX[i] − P0[i]) / Pd[i]
  else
    t_min[i] = (MAX[i] − P0[i]) / Pd[i]
    t_max[i] = (MIN[i] − P0[i]) / Pd[i]
  end if
end for
if t_min[i] > t_max[j] and i ≠ j then
  return false
end if
if t_max[i] < 0 then
  return false
end if
t = mininum(t_min[i])
P = P0 + t · Pd
N = [0, 0, 0]
if P[0] = MIN[0] then
  N[0] = −1
else if P[1] = MIN[1] then
  N[1] = −1
else if P[2] = MIN[2] then
  N[2] = −1
else if P[0] = MAX[0] then
  N[0] = 1
else if P[1] = MAX[1] then
  N[1] = 1
else if P[2] = MAX[2] then
  N[2] = 1
end if
return true
```

spočítat průsečík s paprskem? Tímto se dostáváme k důležitému způsobu optimalizace ray-tracingu, tedy zefektivnění jednotlivých výpočtů průniku paprsku s objektem. Je zřejmé, že čím méně numerických výpočtů, tím lépe. Na dnešních procesorech jsou nejpomalejšími matematickými operacemi dělení, odmocniny a trigonometrické funkce. Operace jako sčítání či násobení jsou až 50 krát rychlejší. Proto jakékoli snížení počtu operací přinese poměrně značné urychlení celého ray-tracingu, když většinu jeho času zabírají právě výpočty průsečíků paprsku s objekty. Optimalizace tohoto druhu je u dnešních renderovacích programů již prakticky samozřejmostí. K popisu principu lepšího způsobu výpočtu využijeme matematickou teorii z oblasti homogenních souřadnic uvedených v kapitole 2.1.1.

Princip je jednoduchý a stejný pro všechny objekty. Předvedeme jej na obecném objektu. Předně si uvědomme, jaké transformace má smysl aplikovat na paprsek s počátkem v bodě P_0 a směrem $\vec{P_d}$. Je zřejmé, že posunutí nemá vůbec význam na vektor $\vec{P_d}$, tedy jej použijeme jen na bod P_0. Jinak všechny ostatní transformace objektu je potřeba aplikovat na obě složky paprsku. Mějme tedy homogenní transformační matici $\mathbf{M}$ objektu. Umíme spočítat i její inverzní matici $\mathbf{M}^{-1}$. Postup je následující:

1. transformujeme paprsek $(P_0, \vec{P_d})$ do lokálních souřadnic objektu její inverzní maticí transformace $\mathbf{M}^{-1}$

$$\begin{aligned} P'_0 &= \mathbf{M}^{-1} \cdot P_0, \\ \vec{P'_d} &= \frac{\mathbf{M}^{-1} \cdot \vec{P_d}}{\|\mathbf{M}^{-1} \cdot \vec{P_d}\|}, \end{aligned}$$

2. vypočítáme průsečík P'_I paprsku $(P'_0, \vec{P'_d})$ s objektem a jeho normálu $\vec{n'}$,

3. aplikujeme transformační matici objektu na P'_I i na $\vec{n'}$ a získáme průsečík P_I s normálou $\vec{n}$ ve světových souřadnicích

$$\begin{aligned} P_I &= \mathbf{M} \cdot P'_I, \\ \vec{n} &= \frac{\mathbf{M} \cdot \vec{n'}}{\|\mathbf{M} \cdot \vec{n'}\|}. \end{aligned}$$

Musíme dát ale obzvláště pozor na transformaci normály. Výše uvedený postup platí pro transformační matici takovou, že $\mathbf{M}^{-1} = \mathbf{M}^T$. Pro normálu totiž musí i po transformaci platit, že je kolmá k tečně $\vec{t'}$ v bodě průniku, tedy

$$\vec{n'} * \vec{t'} = 0.$$

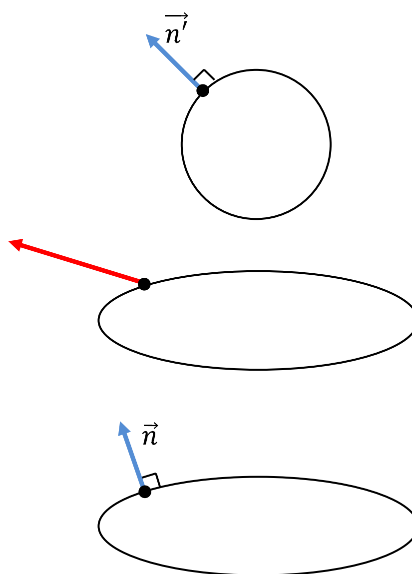
Transformuje-li ale matice $\mathbf{M}$ tečnu $\vec{t'}$ na novou tečnu $\vec{t}$, musí i poté platit, že

$$\vec{n} * \vec{t} = 0.$$

Normálu proto musíme transformovat maticí $\mathbf{N}$ získanou následovně:

$$\begin{aligned} 0 = \vec{n} * \vec{t} &= \vec{n}^T \cdot \vec{t} \\ &= (\mathbf{N} \cdot \vec{n})^T \cdot \mathbf{M} \cdot \vec{t} \\ &= \vec{n}^T \cdot \mathbf{N}^T \cdot \mathbf{M} \cdot \vec{t}. \end{aligned}$$

Což platí, když $\mathbf{N}^T \cdot \mathbf{M} = \mathbf{I}$, a tedy $\mathbf{N}^T = \mathbf{M}^{-1}$. Proto transformační matice normály musí být $\mathbf{N} = (\mathbf{M}^{-1})^T$. Odlišnosti správně a špatně transformovaných normál lze pozorovat na obrázku 2.



Obrázek 2.: **Transformace normály.** Nahoře původní normála. Uprostřed nesprávně transformovaná normála. Dole správně transformovaná normála.

Shrneme-li tento princip výpočtu, zjistíme, že mnoho výpočetních operací navíc nepřinesl, a tedy je velmi výhodné používat jej na všechny objekty, jejichž výpočet v základní poloze je značně jednodušší než v obecné poloze. V našem případě u obecné krychle je rozdíl enormní.

Snížením počtu matematických operací však nemusí být s optimalizací konec. Většina výpočtů totiž disponuje vlastností opakujících se operací pro každý paprsek. Toho se dá využít předvýpočty na paprsku nezávislých operací, což ve výsledku přináší další redukci času výpočtu.

2.4.3. Kužel

Další nový objekt představíme kužel. Jedná se o kvadriku s obecnou rovnicí:

$$\frac{x^2}{a^2} + \frac{y^2}{b^2} = \frac{z^2}{c^2},$$

jehož hlavní osa je podél osy z. Chceme-li mít kužel podél osy y, rovnici upravíme:

$$\frac{x^2}{a^2} + \frac{z^2}{b^2} = \frac{y^2}{c^2}.$$

Výsledkem rovnice jsou ovšem dva kužele, takže je třeba se rozhodnout, který z nich budeme vždy počítat. Složitost výpočtu průsečíku s paprskem je srovnatelná s dřívější kvadrikou – válcem. Kužel máme zadaný jeho vrcholem C, výškou h, poloměrem r, a normalizovanou osou $\vec{a}$. Opět jsou nutné dva výpočty – podstava a plášť.

Začneme pláštěm, který se provede naprosto stejně, jako tomu bylo u válce. Nejprve najdeme střed podstavy B a určíme jeho rovinu s normálou $\vec{a}$ obecnou rovnicí, k níž nám chybí zjistit neznámý parametr D:

$$\begin{aligned} \vec{a} \cdot B + D &= 0, \\ a_x \cdot B_x + a_y \cdot B_y + a_z \cdot B_z + D &= 0. \end{aligned}$$

Při testu, zda paprsek protíná podstavu kužele, zjistíme bod průniku X s rovinou podstavy a ověříme, zda X leží v dosahu poloměru od středu podstavy:

$$\sqrt{(X_x - B_x)^2 + (X_y - B_y)^2 + (X_z - B_z)^2} \leq r^2.$$

Průnik s pláštěm je poměrně složitější. Použijeme označení z obrázku 3. Nechť paprsek p protne plášť válce v bodě Y. Spočítáme vektory

$$\begin{aligned} \vec{v_1} &= Y - C, \\ \vec{v_2} &= (\vec{v_1} * \vec{a}) \cdot \vec{a}, \\ \vec{w} &= \vec{v_1} - \vec{v_2}, \end{aligned}$$

kde vektor $\vec{v_2}$ značí promítnutí vektoru $\vec{v_1}$ na osu $\vec{a}$. Provedeme pozorování. Bod průniku Y leží na plášti kužele, když

$$\begin{aligned} T &= \frac{\|\vec{w}\|}{\|\vec{v_2}\|} = \frac{r}{h}, \\ \vec{w} * \vec{v_2} &= T^2 \cdot (\vec{v_2} * \vec{v_2}). \end{aligned}$$

Platí:

$$\begin{aligned} \vec{v_2} * \vec{v_2} &= (\vec{v_1} * \vec{a})^2, \\ \vec{v_1} * \vec{v_2} &= (\vec{v_1} * \vec{a})^2, \\ \vec{w} * \vec{w} &= \vec{v_1} * \vec{v_1} - (\vec{v_1} * \vec{a})^2. \end{aligned}$$

A tedy z toho vyplývá, že Y leží na plášti, když

$$\begin{aligned} \vec{v_1} * \vec{v_1} - (\vec{v_1} * \vec{a})^2 &= T^2 \cdot (\vec{v_1} * \vec{a})^2, \\ \vec{v_1} * \vec{v_1} &= (1 + T^2) \cdot (\vec{v_1} * \vec{a})^2. \end{aligned} \tag{3}$$

Označme $S = 1+T^2$. Nyní dáme výše zmíněné pozorování dohromady s paprskem v počátku P_0 a směrem $\vec{P_d}$:

$$\begin{aligned} Y &= P_0 + t \cdot \vec{P_d}, \\ \vec{v_1} &= Y - C = (P_0 - C) + t \cdot \vec{P_d} = \vec{p} + t \cdot \vec{P_d}, \end{aligned}$$

kde $\vec{p} = P_0 - C$. Dosazením do rovnice (3) získáme

$$(\vec{p} + t \cdot \vec{P_d}) \cdot (\vec{p} + t \cdot \vec{P_d}) = S \cdot ((\vec{p} + t \cdot \vec{P_d}) * \vec{a})^2. \tag{4}$$

Upravíme levou i pravou stranu rovnice (4):

$$\begin{aligned} (\vec{p} + t \cdot \vec{P_d}) * (\vec{p} + t \cdot \vec{P_d}) &= \vec{p} * \vec{p} + 2t \cdot (\vec{p} * \vec{P_d}) + t^2 \cdot (\vec{P_d} * \vec{P_d}), \\ S \cdot ((\vec{p} + t \cdot \vec{P_d}) * \vec{a})^2 &= S \cdot ((\vec{p} * \vec{a})^2 + 2t \cdot (\vec{p} * \vec{a}) \cdot (\vec{P_d} * \vec{a}) + t^2 \cdot (\vec{P_d} * \vec{a})^2) \end{aligned}$$

a přesuneme na jednu stranu:

$$\begin{aligned} 0 &= t^2 \cdot (\vec{P_d} * \vec{P_d} - S \cdot (\vec{P_d} * \vec{a})^2)) + 2t \cdot (\vec{p} * \vec{P_d} - S \cdot (\vec{p} * \vec{a}) \cdot (\vec{P_d} * \vec{a})) + \\ &\quad + (\vec{p} * \vec{p} - S \cdot (\vec{p} * \vec{a})^2). \end{aligned}$$

Což je kvadradická rovnice:

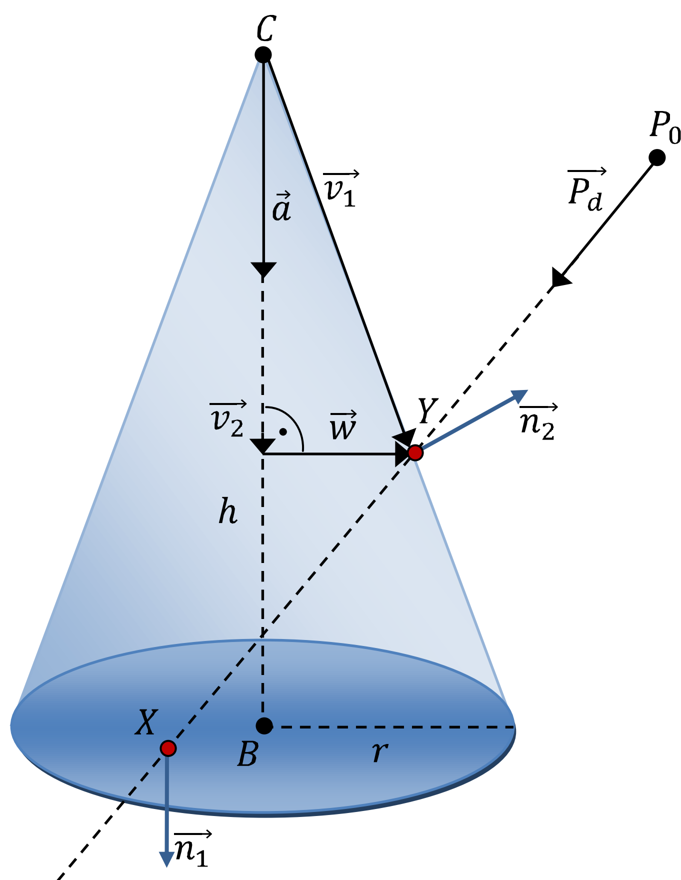
$$\begin{aligned} 0 &= a \cdot t^2 + b \cdot t + c, \\ a &= \vec{P_d} * \vec{P_d} - S \cdot (\vec{P_d} * \vec{a})^2, \\ b &= 2 \cdot (\vec{p} * \vec{P_d} - S \cdot (\vec{p} * \vec{a}) \cdot (\vec{P_d} * \vec{a})), \\ c &= \vec{p} * \vec{p} - S \cdot (\vec{p} * \vec{a})^2, \\ d &= b^2 - 4 \cdot a \cdot c, \\ t_{1,2} &= \frac{-b \pm \sqrt{d}}{2 \cdot a}. \end{aligned}$$

Vyjde-li d záporné, tak paprsek kužel neprotíná. Jinak vybereme menší kladné t_i a vypočítáme bod průniku $Y = P_0 + t_i \cdot \vec{Pd}$. Zbývá ještě určit normálu. Využijeme již spočítaný bod Y:

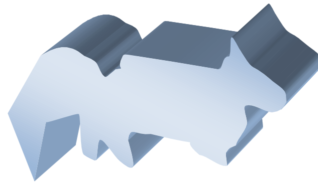
$$\begin{aligned} \vec{v_1} &= Y - C, \\ \vec{v_2} &= (\vec{v_1} * \vec{a}) \cdot \vec{a}, \\ \vec{w} &= \vec{v_1} - \vec{v_2}, \\ \vec{n_2} &= (\vec{a} \times \vec{w}) \times \vec{v_1}. \end{aligned}$$

Nakonec je nutno rozhodnout, zda se jedná o správný kužel. Tedy musí mít maximálně výšku h a ležet v kladné polorovině:

$$\begin{aligned} \|\vec{v_2}\| &\leq h, \\ \vec{v_1} * \vec{a} &\geq 0. \end{aligned}$$



Obrázek 3.: **Průsečík paprsku s kuželem.**



Obrázek 4.: **Obecný objekt.**

2.4.4. Obecný objekt

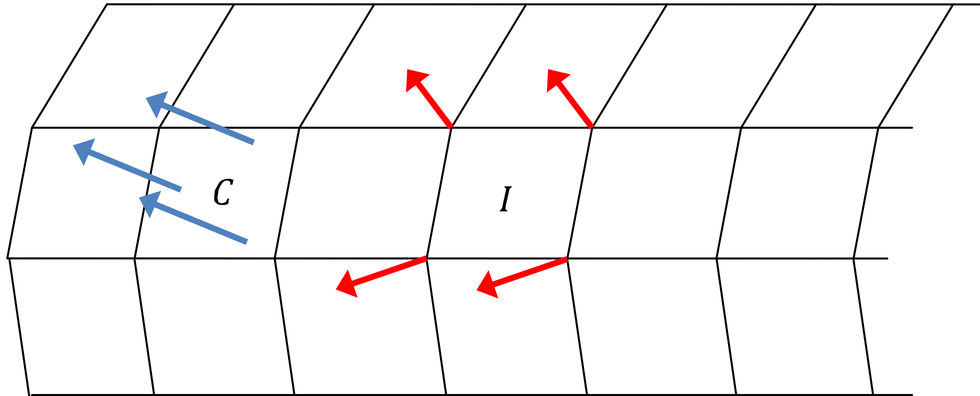
Všechny dosud představené objekty vyplývající většinou z jejich implicitní rovnice mají jednu společnou neblahou vlastnost – jsou téměř nepoužitelné v moderní praktické počítačové grafice. Ta nepoužitelnost spočívá v tom, že i kdybychom se sebevíc snažili tyto objekty různě zkombinovat, nikdy bychom nevytvořili libovolně vypadající objekt. Např. objekt z obrázku 4. bychom těžko zvládli vymodelovat se všemi detaily. K tomu je zapotřebí jednodušších objektů jako jsou polygony. Nejčastěji používaným polygonem je trojúhelník, ale některé grafické enginy využívají i jiné typy. Dnešní grafické karty s těmito polygony umí velmi efektivně pracovat. Důraz se klade na vlastnosti tohoto polygonu – co nejhladší možná interpolace normál a samozřejmě složitost výpočtu průniku s paprskem.

2.4.5. Trojúhelníková ploška

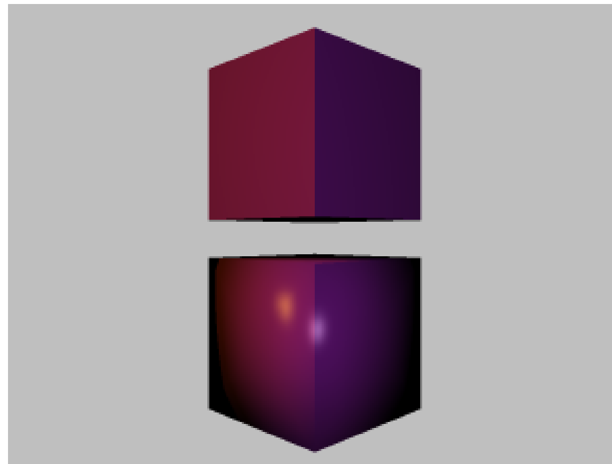
Představíme si výpočet průniku s trojúhelníkovou ploškou pomocí barycentrických souřadnic, jenž kromě lepší časové složitosti poskytuje i lepší (přirozenější) interpolaci normál sousedních plošek. Tento způsob interpolace je implementací principu zvaného *Phongova interpolace normál* a je velmi rozšířený vzhledem k jeho jednoduchosti (např. v OpenGL). Nicméně mohou nastat případy viditelných artefaktů, a tedy existuje celá řada různých variací kvalitnějšího výpočtu normál, avšak jejich přínos je doprovázen náročnějším výpočtem, a tedy nejsou příliš rozšířené.

Přednější je ale srovnání jednotně počítané normály pro celou plošku s normálou interpolovanou (viz obrázek 5.). Předpokládejme, že plošky C a I mají odlišný způsob výpočtu normál. U plošky C bude mít modrý vektor stále stejný směr, ať jej budeme počítat v kterémkoli bodě uvnitř polygonu. Je zřejmé, že mezi sousedními ploškami nebude žádný plynulý přechod, ale skoková změna. Naproti tomu uvnitř plošky I interpolací červených vektorů – kdy bod průniku blíže šipce bude šipkou i více ovlivněn – dosáhneme v každém bodě uvnitř polygonu jiné normály. Praktický rozdíl lze pozorovat na obrázku 6., detailněji viz obrázek 7.

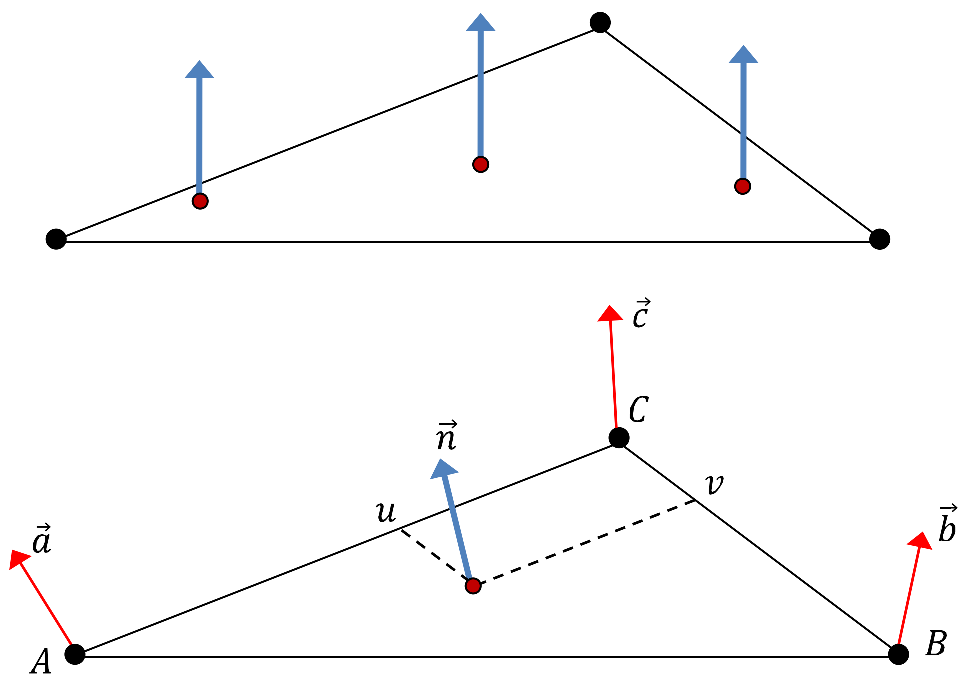
Nyní popíšeme způsob výpočtu interpolované normály (pro ilustraci viz obrázek 8.). Mějme bod X, ve kterém chceme počítat normálu (např. průsečík



Obrázek 5.: **Normály plošek objektu.** Normály plošky C jsou konstantní v každém bodě plošky. Normály plošky I jsou interpolovány normálami v hraničních bodech.



Obrázek 6.: **Ukázka interpolace normál.** Nahoře konstantní výpočet normál pro každý bod plošky. Dole Phongova interpolace normál. Ukázka z programu RayTracer, testovací balík: *Interpolace.*



Obrázek 7.: **Způsoby výpočtů normál.** Nahoře konstantní normála v každém bodě plošky. Dole normála v bodě průniku interpolována normálami hraničních bodů pomocí barycentrických souřadnic.

paprsku s ploškou). Proložíme bod libovolnou přímkou tak, aby prošla dvěma hraničními přímkami plošky. Na každé hraniční přímce leží právě dva hraniční body, jejichž normála se nemění. Tyto dvě normály lineárně interpolujeme v hraničním bodě průniku. Vzniknou tak dvě interpolované normály $\vec{n_{1,3}}$ (interpolací normál $\vec{n_1}$ a $\vec{n_3}$) a $\vec{n_{2,3}}$ (interpolací normál $\vec{n_2}$ a $\vec{n_3}$) na hraničních přímkách. Zbývá provést opět lineární interpolaci normál $\vec{n_{1,3}}$ a $\vec{n_{2,3}}$ k dosažení výsledné normály $\vec{n}$. Každá normála je při výpočtu normalizována.

Již jsme uvedli, že k výpočtu průniku paprsku s ploškou využijeme barycentrických souřadnic. Jejich výhoda spočívá v efektivnosti výpočtu u ověření, zda bod leží uvnitř trojúhelníku. Nejdříve ale uvedeme konstantní výpočet normál, jenž se nám bude také později hodit. Mějme trojúhelníkovou plošku zadanou třemi body A, B a C. Normálu spočteme jednoduchým, dobře známým způsobem:

$$\vec{n} \;=\; (A - C) \times (B - C).$$

Záměrně je uveden zápis s bodem C jako tzv. počáteční bod. Při testování, zda bod průniku X s rovinou trojúhelníku leží i uvnitř trojúhelníku, můžeme použít tento výpočet pro postupně různé počáteční body A, B, C, X a porovnávat při tom znaménka normál – zda normála ukazuje vždy na stejnou stranu. Detaily zde nebudeme uvádět proto, že tento způsob je značně neefektivní.

Využijeme raději *barycentrických souřadnic*. Víme, že tři body již tvoří rovinu. Označíme-li jeden z těchto bodů jako počátek, ostatní body roviny můžeme určit relativně vzhledem k tomuto bodu. Počátkem trojúhelníku prochází dvě přímky. Urazíme-li nějakou vzdálenost po jedné z nich a odtud pak další vzdálenost po druhé z nich, můžeme se dostat do libovolného bodu na rovině. Matematicky lze tyto body popsat jako

$$P \;=\; C + u \cdot (A - C) + v \cdot (B - C),$$

kde parametry $u \in \mathbb{R}$ a $v \in \mathbb{R}$ se nazývají barycentrické souřadnice. Všechny body uvnitř trojúhelníku těmito souřadnicemi popíšeme jako dvojici (u, v):

$$\begin{aligned} u &> 0, \\ v &> 0, \\ u + v &< 1. \end{aligned}$$

Barycentrické souřadnice trojúhelníku se někdy uvádí i jako trojice (u, v, w), kde

$$\begin{aligned} u + v + w &= 1, \\ w &= 1 - u - v. \end{aligned}$$

Pak každý bod uvnitř trojúhelníku lze pak popsat jako

$$P \;=\; u \cdot A + v \cdot B + w \cdot C. \tag{5}$$

Jediné, co nám zbývá vyjasnit, je, jak tyto souřadnice vypočítáme. Opět, trojúhelník bude tvořen třemi body A, B, C a máme spočítaný průsečík X paprsku s rovinou trojúhelníku. Vypočítáme tři hlavní vektory

$$\begin{aligned} \vec{v_0} &= A - C, \\ \vec{v_1} &= B - C, \\ \vec{v_2} &= X - C \end{aligned}$$

a z rovnice

$$X = C + u \cdot (A - C) + v \cdot (B - C)$$

získáme jednu rovnici o dvou neznámých:

$$\vec{v_2} = u \cdot \vec{v_0} + v \cdot \vec{v_1}. \tag{6}$$

Pomůžeme si vytvořením dvou rovnic. První rovnici získáme skalátním vynásobením obou stran rovnice (6) vektorem $\vec{v_0}$ a druhou rovnici získáme skalárním vynásobením rovnice (6) vektorem $\vec{v_1}$. Nové rovnice pak budou tvaru:

$$\begin{aligned} \vec{v_2} * \vec{v_0} &= u \cdot (\vec{v_0} * \vec{v_0}) + v \cdot (\vec{v_1} * \vec{v_0}), \\ \vec{v_2} * \vec{v_1} &= u \cdot (\vec{v_0} * \vec{v_1}) + v \cdot (\vec{v_1} * \vec{v_1}). \end{aligned}$$

Nyní už máme dvě rovnice o dvou neznámých, což umíme řešit například touto posloupností výpočtů:

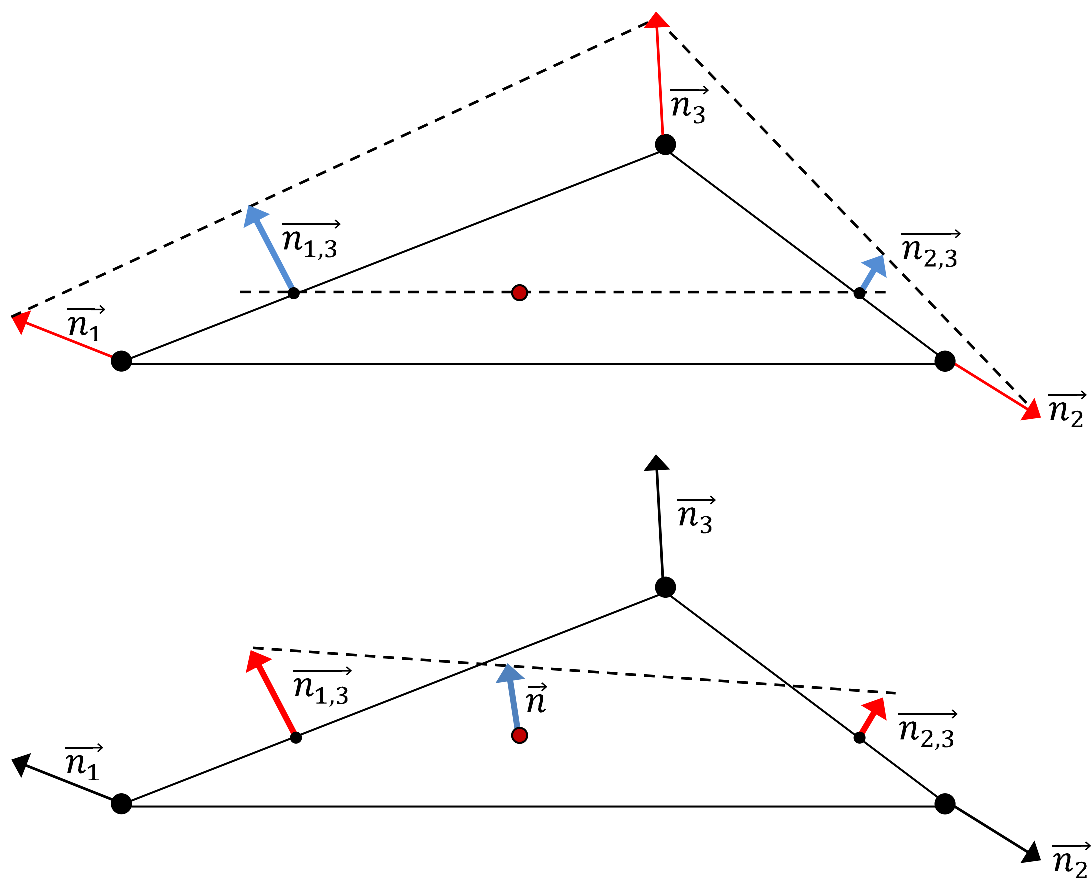
$$\begin{aligned} \vec{v_{00}} &= \vec{v_0} * \vec{v_0}, \\ \vec{v_{01}} &= \vec{v_0} * \vec{v_1}, \\ \vec{v_{02}} &= \vec{v_0} * \vec{v_2}, \\ \vec{v_{11}} &= \vec{v_1} * \vec{v_1}, \\ \vec{v_{12}} &= \vec{v_1} * \vec{v_2}, \\ d &= \vec{v_{00}} * \vec{v_{11}} - \vec{v_{01}} * \vec{v_{01}}. \end{aligned}$$

Konečně vypočítáme souřadnice u, v a w jako

$$\begin{aligned} u &= \frac{\vec{v_{11}} * \vec{v_{02}} - \vec{v_{01}} * \vec{v_{12}}}{d}, \\ v &= \frac{\vec{v_{00}} * \vec{v_{12}} - \vec{v_{01}} * \vec{v_{02}}}{d}, \\ w &= 1 - u - v. \end{aligned}$$

Je-li bod obsažen v trojúhelníku, vypočítáme i normálu v průsečíku z normál obsažených ve vrcholech trojúhelníku:

$$\vec{n} = u \cdot \vec{a} + v \cdot \vec{b} + w \cdot \vec{c}. \tag{7}$$



Obrázek 8.: **Interpolace normál.** Nahoře výpočet interpolovaných krajních normál $\vec{n_{1,3}}$ a $\vec{n_{2,3}}$ na hranách plošky. Dole výsledná normála $\vec{n}$ v bodě průniku interpolovaná krajními dvěma normálami.

Poslední z maličkostí, která nám chybí doplnit, je výpočet vrcholových normál. K tomu ale potřebujeme doplnit informace o reprezentaci trojúhelníku. V rovnici 7 a na obrázku 7. používáme normály, které jsme ale trojúhelníku nezadali. Půjdeme na okamžik do teoretické reprezentace objektu pomocí trojúhelníkových plošek. Existuje několik způsobů, jak tento objekt reprezentovat.

1. **Síť vrcholů**
   Každý vrchol obsahuje kromě svých souřadnic i odkaz na všechny vrcholy, s kterými leží na společné hraně. Tento přístup je sice nejjednodušší, ale málo používaný, jelikož většina informací o topologii objektu se nedá zjistit přímo. Výhoda například spočívá v nízké paměťové náročnosti, takže i při určitých aplikací má uplatnění.

2. **Seznam vrcholů a plošek**
   Přirozenější reprezentace hlavně pro rendering, kdy je objekt reprezentován dvěma seznamy. První obsahuje seznam plošek s údaji o vrcholech tvořících každou z nich. Druhý seznam je naopak zaplněn vrcholy, u nichž je přítomen údaj o všech ploškách které ohraničuje. Tímto způsobem reprezentace můžeme rovnou explicitně procházet plošky objektu bez potřeby je pracným výpočtem zjišťovat. Dnešní grafické karty navíc poskytují rozhraní přijímající rovnou vykreslované plošky, neboli jejich vrcholy. Tuto reprezentaci jsme zvolili v implementaci obecného objektu v programu RayTracer a na ní si vysvětlíme zbývající výpočty u plošky.

3. **Síť okřídlených hran**
   Složitější struktura, která kromě seznamu plošek a vrcholů obsahuje i seznam hran. Poskytuje širokou škálu operací, které se s ní dají rychle provádět a proto se jedná o velmi flexibilní a rozšířenou strukturu vhodnou jak pro rendering, tak pro interaktivní aplikace. Nevýhodou jsou však vyšší paměťové nároky.

Důležitá je pak množina operací, které se s těmito strukturami dají vykonávat v krátkém čase. Uvedeme jen základní, často potřebné pro rendering. Jedná se např. o

- nalezení všech vrcholů plošky,
- nalezení všech sousedních vrcholů k vrcholu,
- nalezení všech plošek k vrcholu,
- nalezení všech hran k vrcholu.

Nyní, když víme, jakou máme strukturu a jaké operace po ní můžeme požadovat, vrátíme se k výpočtu normály plošky. Při pohledu na obrázek 7. vidíme normály $\vec{a}$, $\vec{b}$, $\vec{c}$ vrcholů trojúhelníku. Otázka zní, jak je zjistit, máme-li zadány pouze

souřadnice těchto bodů. Postup je následující. Nejprve se spočítá normála plošky z těchto tří bodů (statická normála stejná pro každý bod uvnitř trojúhelníku – viz rovnice (5)). Tuto normálu vypočítáme pro každou plošku a zapamatujeme si ji. Následně, díky námi vhodně zvolené reprezentaci objektu, můžeme u každého vrcholu iterovat přes všechny plošky, k nimž náleží, a počítat z nich normálu vrcholu jako součet normál jeho plošek. Výslednou normálu ještě normalizujeme. Tím získáme průměrnou normálu všech okolních plošek.

Ještě pro úplnost uvedeme rozumně efektivní výpočet průsečíku X s paprskem v počátku P_0 a směru $\vec{P_d}$. Máme-li zapamatovánu statickou normálu $\vec{n}$ plošky, tak bod průniku X určíme jako

$$\begin{aligned} k_1 &= \vec{n} * (C - P_0), \\ k_2 &= \vec{n} * \vec{P_d}, \\ t &= \frac{k_1}{k_2}, \\ X &= P_0 + t \cdot \vec{P_d}. \end{aligned}$$

Samozřejmě musí platit, že $k_2 \neq 0$ a $t > 0$.

3. Optimalizace ray-tracingu

Nyní se dostáváme k hlavní části práce, a tedy k optimalizacím algoritmu ray-tracing. Nejprve zopakujeme naivní algoritmus a představíme jeho neefektivní časovou složitost. V kapitole 3.2. uvedeme základní principy optimalizace a přístupy k porovnávání výpočtů ray-tracingu. Ve zbývajících kapitolách postupně probereme jednotlivé metody optimalizace, předvedeme jejich výsledky a pokusíme se je porovnat. Jako motivaci do dalších kapitol a užitečnosti optimalizací ray-tracingu viz obrázek 9. Jestliže trvalo vytvoření tohoto obrázku za použití optimalizačních technik 73 hodin, kolik času by asi trval původní naivní algoritmus?

Obrázek 9.: **Model Australské Královské Koruny** vytvořen programem *Blender* a vygenerován programem *LuxRender*. Ve scéně je přibližně $1,8$ milionu vrcholů, osvětlení tvoří 6 plošných světel a hustota vzorkování je 1280 vzorků na pixel. Výpočet (rok 2009) trval 73 hodin s 64bitovým procesorem quad-core o využité paměti přibližně $6,7$ GB. Pro více informací o autorovi M. Lubichovi a jeho projektu viz www.loramel.net.

3.1. Ray-tracing

Metoda sledování paprsku je fyzikální technika z 19. století na bázi optických čoček. Řídí se zákony fyziky jako je odraz, lom a šíření světla. Algoritmus ray-tracing je postaven přesně na této bázi. Stručně si jej připomeneme.

Jedná se o metodu *zpětného sledování paprsku*, jelikož nevypouštíme paprsky od světelných zdrojů, ale od pozorovatele. Generátor paprsků – kamera – produkuje paprsky, jejichž interakci ve scéně zkoumáme. Pokaždé, když kamera vypustí paprsek, musíme zjistit, který objekt protíná a v jakém bodě. Průsečíků objektů s paprskem může být velmi mnoho, ale nás z nich zajímá pouze ten nejbližší a normála v něm. K tomu nám poslouží vědomosti získané z kapitoly 2.

Rozlišujeme dva druhy paprsků – primární a sekundární. Primární je každý paprsek vycházející z kamery před prvním průnikem s objektem. Na základě odrazivých vlastností materiálu objektu se může paprsek odrazit, zlomit nebo absorbovat. Klidně i všechno současně (můžeme konstruovat jakékoli – i nereálné – materiály). Odrazem i lomem vznikají sekundární paprsky, které budeme opět rekurzivně sledovat dalším průchodem scény. Aby proces neběžel do nekonečna, stanovuje se maximální povolená hloubka rekurze – většinou 3 až 5. Navíc, v každém bodě průsečíku se zjistí tzv. vržený stín a pro každé nezastíněné světlo se spočítá lokální barva, jež přispěje do celkové barvy součtem s barvami nesenými sekundárními paprsky. Nakonec se barva pixelu určí jako součet všech barev nesených na všech paprscích vzniklých z primárního paprsku. Algoritmus 3.1. převzatý z [2] tento postup shrnuje. Detaily algoritmu se již více nebudeme zabývat a odkazujeme buď na předešlou práci [2] nebo třeba na [5, 10, 13].

Algoritmus 3.1. Renderování obrázku ray-tracingem

Vstup: rozměry obrázku, maximální hloubka rekurze
Výstup: barvy všech pixelů obrázku
Cyklus přes všechny pixely:

1. pro daný pixel obrázku vypočti z generátoru paprsků paprsek *in*,
2. urči barvu pixelu metodou *Raytrace(in, 0)*.

Než se pustíme do optimalizace, řekneme si něco o složitosti tohoto základního algoritmu. Říká se mu *naivní* nebo algoritmus *hrubé síly*, jelikož vždy – při každém hledání průsečíku objektu s paprskem – se prochází **všechny** objekty ve scéně. I když je tento přístup správný, je velmi pomalý. Máme-li ve scéně N objektů a obrázek obsahuje I pixelů, tak časová složitost algoritmu pro primární paprsky je $O(I \cdot N)$. Obecně pro počet P všech paprsků vygenerovaných během výpočtu algoritmus běží v čase $O(P \cdot N)$. Pro složité scény s tisíci až miliony objekty je tato složitost nepřijatelná. V následující kapitole si představíme metody a struktury, jak lineární složitost snížit víceméně na logaritmickou. Logaritmická složitost je totiž dosažena jen v průměrném případě, jelikož optimalizovaný algoritmus bývá doprovázen náročnými předvýpočty, složitými strukturami a doplňujícími jednoduššími výpočty. Existují však algoritmy, u nichž je logaritmická časová složitost zaručena, avšak ty pracují jen s určitými druhy scén.

Algoritmus 3.2. Ray-tracing

Vstup: sledovaný paprsek in, aktuální hloubka rekurze h
Výstup: barva paprsku

Raytrace(paprsek in, hloubka h):

1. vypočti průsečík P paprsku in s nejbližším objektem ve scéně,
   (a) není-li průsečík, vrať barvu pozadí scény;
2. pomocí stínových paprsků zjisti všechny světelné zdroje L_i přímo osvětlující průsečík P,
3. ze všech L_i vypočti barvu b_1 lokálním osvětlovacím modelem v bodě P,
4. jestliže hloubka h nepřekročila maximální hloubku rekurze:
   (a) připouští-li to materiál, spočti odražený paprsek R_r a vyšli ho: $b_2 = Raytrace(R_r, h + 1)$
   (b) připouští-li to materiál, spočti lomený paprsek R_t a vyšli ho: $b_3 = Raytrace(R_t, h + 1)$
   (c) přičti k barvě b_1 také b_2 a b_3: $b_1 = b_1 + b_2 + b_3$
5. vrať barvu b_1

3.2. Optimalizační principy

„Premature optimization is the root of all evil."
– DONALD. E. KNUTH [8]

Optimalizační metody fotorealistického zobrazování by se v dnešní době daly nazvat metodami základními, protože původní naivní principy použité v počátcích této disciplíny jsou pro rozsáhlé scény prakticky nepoužitelné vzhledem k jejich časové složitosti. V současné době není již požadavek na fotorealistický výsledek nadřazen nad celkový čas, jelikož čím dál častěji je požadován výstup v reálném čase a fotorealistická kvalita se stala standardem. S naivním přístupem algoritmu ray-tracingu procházením všech objektů ke každému paprsku je tento úkol nesplnitelný. Samozřejmě, že rok od roku s výkonnějšími procesory dosahujeme výstupu rychleji, ale pořád tento přístup není schopen poskytnout přijatelné výsledky v rozumném čase (ani nikdy nebude). Vždy totiž bude požadavek náročnějších scén s ještě složitějšími vizuálními efekty. I když první průkopníci hlavní optimalizační techniky založené na metodě *BVH (bounding volume hierarchy)*[1] Kay a Kajiya v roce 1986 hájili ray-tracing, že není pomalý, ale že pomalé jsou pouze počítače, navrhli jeho vskutku přelomové urychlení. Všechny následující techniky z jejich práce víceméně vycházejí a i když mají také jisté kvalitní výsledky, zlepšení už není tak markantní.

3.2.1. Porovnávání optimalizačních metod

V této kapitole se pokusíme uvést hlavní metody optimalizace založené na akcelerujících strukturách. Metody vždy podrobně popíšeme a pokusíme se je srovnat jak s naivním přístupem, tak mezi sebou. Příklad porovnání je uveden na obrázku 10. a tabulce 1. Porovnávání však s sebou nese značná úskalí, s nimiž se grafici potýkají již od vzniku ray-tracingu. Složitost algoritmu je totiž ovlivněna spoustou různých aspektů a porovnávat například přesný čas mezi různými testovacími prostředími je zcela bezvýznamné. Některé implementace se zaměřovaly na malé scény a obsahovaly specifickou reprezentaci objektů, jiné byly optimalizovány třeba na vržené měkké stíny apod. Vznikla potřeba standardizace grafických prostředí, aby se zjistilo, jak si který optimalizační přístup vede v porovnání s ostatními. Obzvláště na konci 80. let minulého století s rozvojem optimalizačních technik bylo nutné je mezi sebou efektivně porovnat. V roce 1987 vznikla veřejná a dodnes používaná databáze standardních grafických prostředí [6], obsahující detailní popis scény s informacemi o osvětlení, kameře, polygonech, normálách v bodech atd. Pro vlastní testování je potřeba převést tyto informace

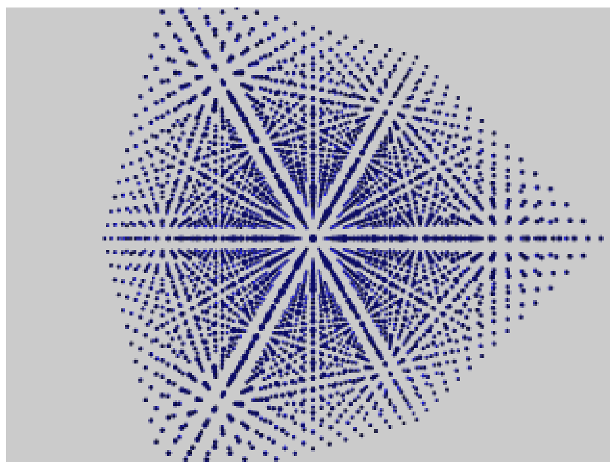
[1]Poznámka. Termín *bounding volume* je již v počítačové grafice tak zažitý, že i v tomto textu upustíme od jeho nepřesného překladu *obálka* (použitého např. v [13]) a budeme používat většinou originální termín.

do svého konkrétního modelu. Kromě algoritmů ray-tracingu se dají takto testovat i hardwarové grafické akcelerátory. Ray-tracing se hned zpočátku doporučil neporovnávat přesným časem kvůli různým architekturám a programovacím jazykům. Proto grafici přišli s myšlenkou porovnávat algoritmy raději počtem výpočtů průsečíků paprsku s objekty.

Jelikož software RayTracer je napsán v jazyce C#, je i zde naprosto zbytečné porovnávat výpočty mezi sebou časem vzhledem ke správci paměti. I dva stejné výpočty se někdy při testování lišily až o půl sekundy (dokonce i při ručním zavoláním správce paměti před začátkem výpočtu). Délku výpočtu může totiž nemálo ovlivnit i jakákoli vedlejší činnost operačního systému. Navíc je ověřeno, že minimalizace editoru během výpočtu také razantně sníží celkový čas. V následujících kapitolách proto nebudeme vůbec zvažovat jako hlavní porovnávací metriku čas – občas jej zmíníme jen jako doplnění k demonstraci řádových odlišností metod. Pod každým obrázkem vždy uvedeme i balík, v němž je scéna obsažena na přiloženém DVD – pro vlastní ověření či experimentování. Většinou je v balíku navíc přítomna i animace pro úplnou představu scény. Testovací data byla pořízena na 64bitové architektuře 2jádrového Intel Core™ i5-2 430M CPU @ 2.40 GHz s 8 GB pamětí.

technika optimalizace	**počet výpočtů s objektem**	**počet výpočtů s obálkou**
naivní	$3\,169\,098 \cdot 10^3$	–
Kd-tree	$699 \cdot 10^3$	$61\,904 \cdot 10^3$

Tabulka 1.: **Porovnání naivního principu s optimalizací k obrázku 10.**



Obrázek 10.: **Porovnání naivního principu s optimalizací.** Scéna obsahuje 3 375 koulí (celkově tvořících krychli) a dvě bodová světla. Rozlišení obrázku je 320×240 s hloubkou rekurze 0 a antialiasingem. Pomocí akcelerující struktury kd-tree bylo provedeno $699 \cdot 10^3$ výpočtů průsečíků s koulí a $61\ 904 \cdot 10^3$ výpočtů s obálkou. Bez použití optimalizace bylo provedeno $3\ 169\ 098 \cdot 10^3$ výpočtů průsečíků paprsku s koulí a s časem výpočtu přibližně 10krát delším. Data viz tabulka 1. Balík: *sphericCube*.

3.3. Optimalizace kódu

> *There is no doubt that the grail of efficiency leads to abuse. Programmers waste enormous amounts of time thinking about, or worrying about, the speed of noncritical parts of their programs, and these attempts at efficiency actually have a strong negative impact when debugging and maintenance are considered. We should forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil.*
> – Donald. E. Knuth [1]

Optimalizace na nižších úrovních programování nepatří do oblasti zájmů této práce. Vždy se dá totiž program tímto způsobem ještě více vylepšit. Tyto techniky přímo nepatří ani do počítačové grafiky, nýbrž do počítačové vědy obecně. Tedy nejsou ani považovány za optimalizace algoritmů ray-tracingu. Navíc jejich urychlení je víceméně jen chirurgické, i když na druhou stranu každé zlepšení při 73 hodinovém renderingu je příhodné. Jejich nevýhodou je často znepřehlednění kódu a i z tohoto důvodu jsme v programu tyto optimalizační techniky nezvažovali.

Další urychlení spočívá v efektivní implementaci datových struktur. Jednak jejich vhodnou velikostí, aby se mohl používaný objekt celý načíst do cache paměti

[1] Uvádíme zde celý odstavec z [8], který do kontextu této podkapitoly ještě více zapadá.

a nebylo nutné často přistupovat během výpočtu do pomalejší operační paměti. Dále vhodným indexováním objektů v paměti zvýšíme také efektivitu práce se strukturami. Podrobněji tyto principy nebudeme rozebírat, odkážeme se pouze na [10]. Patří sem také otázka paralelizace výpočtu, ale zde si musíme dát pozor, aby nekolidovala s výše zmíněnou optimalizací. Například sdílením ukazatelů na objekty se snadno můžeme dostat buď do komplikovaně detekovaných potíží, anebo budeme nuceni snížit efektivitu optimalizace datových struktur.

3.4. Akcelerující struktury

V této kapitole si předvedeme moderní přístupy k optimalizaci ray-tracingu. Požadavky na ně jsou zřejmé: ve scéně s n objekty zredukovat lineární časovou složitost $O(n)$ při výpočtu průsečíku paprsku naivního algoritmu na logaritmickou (v lepším případě), prakticky však na složitost sub-lineární. Tyto techniky budou vycházet z předpokladů, že pro většinu paprsků platí, že každý z nich prochází jen velmi malým procentem objektů ve scéně. Když dokážeme rychle zjistit, které objekty může paprsek protnout, a které jistě neprotne, můžeme výrazně snížit počet výpočtů průsečíků. Všechny objekty potřebujeme nějakým způsobem uspořádat na základě jejich rozložení ve scéně. Technikám, které takto s objekty pracují, říkáme *akcelerující techniky* a jejich používané struktury označujeme jako *akcelerující struktury*.

Akcelerující struktury jsou dnes základní komponentou každého ray-tracingu. Jejich cílem je rychlé zamítnutí prohledávaných objektů v procesu výpočtu průsečíku s objekty. Existují různá dělení těchto struktur podle různých vlastností: složitost konstrukce, složitost struktury, paměťová náročnost atd. Každá má samozřejmě své výhody a praktické uplatnění. Obecně by se daly akcelerující struktury rozdělit do dvou hlavních kategorií:

- prostorové rozdělení,
- objektové rozdělení.

Prostorové rozdělení rozkládá prostor scény do většinou disjunktních oblastí a zaznamenává, který objekt scény je překrývá. Existují různá kritéria na oblasti obsahující objekty (např. maximální počet objektů v jedné oblasti, minimální velikost oblasti). Podle nich se může oblast dále rozdělit na menší podoblasti. Je zde jasně vidět rekurzivní konstrukce struktury. Když potom hledáme průsečík objektu s paprskem, nejprve zjistíme, které oblasti paprsek protíná a až poté testujeme všechny jejich přidružené objekty. Zřejmě je efektivita algoritmu značně ovlivněna složitostí výpočtu průsečíku paprsku s oblastí, a tedy oblast by měla být v tomto smyslu co nejjednodušší. Využijeme-li poznatky z kapitoly o objektu AABB (viz kapitola 2.4.1.), získáme jednu z nejčastějších reprezentací 3D oblasti. Na AABB jsou postaveny např. struktury *mřížka*, *octree*, *kd-tree*. Jiný způsob dělení prostoru používá *BSP-tree*.

Objektové rozdělení pracuje lehce odlišným způsobem než dělení prostoru. Podíváme-li se na celou scénu jako na jeden objekt, zjistíme, že jej můžeme rozdělit na menší podobjekty (množiny objektů). Obsahuje-li množina více objektů, opět ji lze rozdělit. Každé skupině objektů (i k jednomu objektu) přitom přiřadíme tzv. *bouding volume*, což je jejich konvexní obálka ve smyslu co nejtěsnějšího ohraničení všech objektů uvnitř (detailněji viz kapitola 3.4.1.). Uspořádáním obálek do nějaké hierarchie můžeme dosáhnout značného zlepšení oproti naivnímu přístupu (podrobněji viz kapitola 3.4.2.). Opět, složitost akcelerující techniky je závislá jednak na složitosti obálky (resp. složitosti výpočtu jejího průsečíku s paprskem), jednak na zvolené hierarchické struktuře.

Mezi oběma přístupy je jeden podstatný rozdíl. Prostorové dělení dělí prostor na disjunktní oblasti, zatímco objektové dělení rozděluje na disjunktní části objekty scény. Oba zmíněné přístupy jsou na poli optimalizačních technik ray-tracingu velmi úspěšné. Není obecný rozhodující důvod k preferování jednoho přístupu před druhým, přesto v kapitole 3.4.8. zkusíme provést porovnání některých metod na specifických scénách.

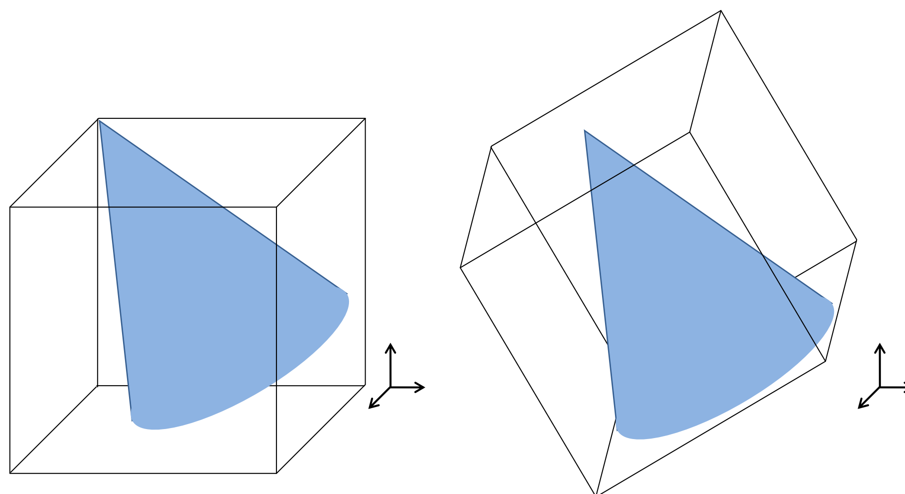
3.4.1. Obálka (BV)

Obálka (anlgicky *bounding volume*) je velmi významný objekt z optimalizačních technik ray-tracingu. Již jsme na jednu obálku narazili (v kapitole 2.4.1.) a ukázali jsme si její jednoduchý výpočet průsečíku s paprskem. Nejjednodušší možnou obálkou je totiž právě AABB, ale na rozdíl od metod dělících prostor, se často u objektového dělené využívají i jiné obálky. Hlavní požadavek je, že se musí jednat o konvexní obal. Další dva požadavky jsou vzájemně v rozporu:

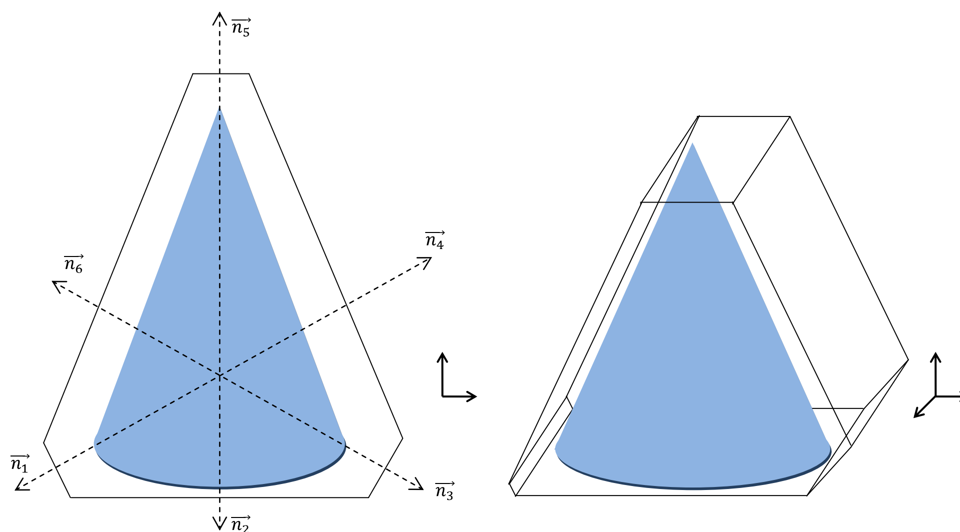
- jednoduchý výpočet průsečíku s obálkou,
- přiléhavost obálky.

Čím více totiž chceme jednodušší výpočet průsečíku, tím více máme jednodušší strukturu obálky, a tedy bohužel menší přiléhavost k objektu. V 80. letech rozvojem optimalizačních technik vzniklo několik návrhů obálek. Kromě klasické AABB například také OBB (*oriented bounding box* – libovolně orientovaný kvádr), sférickou, nebo ke každému objektu vytvořenou na míru (viz obrázky 11. a 12.). Avšak největšího úspěchu dosáhla obálka od Kay a Kajiya popsaná v [7]. Tvůrci poměrně efektivně zkombinovali oba rozporné faktory nezávislým jednotným předdefinováním normál hraničních rovin. Takové omezení má své nevýhody ve flexibilitě, ale naopak lze každou obálku popsat jen několika čísly – normálami. Normály jsou dvojího druhu. První z nich tvoří roviny z AABB. Pro 2D se jedná o normály

$$\begin{pmatrix} 1 \\ 0 \end{pmatrix}, \begin{pmatrix} 0 \\ 1 \end{pmatrix}$$



Obrázek 11.: **Typy obálek 1.** Vlevo souřadnicově orientovaná (AABB – axis-aligned bounding box), vpravo libovolně orientovaná (OBB – oriented bounding box).



Obrázek 12.: **Typy obálek 2.**

a pro 3D o normály

$$\begin{pmatrix}1\\0\\0\end{pmatrix}, \begin{pmatrix}0\\1\\0\end{pmatrix}, \begin{pmatrix}0\\0\\1\end{pmatrix}.$$

Druhá skupina normál tvoří v případě 2D čtyřstěn. Jsou to normály

$$\begin{pmatrix}\frac{\sqrt{2}}{2}\\\frac{\sqrt{2}}{2}\end{pmatrix}, \begin{pmatrix}\frac{\sqrt{2}}{2}\\-\frac{\sqrt{2}}{2}\end{pmatrix}.$$

Ve 3D jsou tyto normály čtyři, definují osmistěn a jejich tvar je

$$\begin{pmatrix}\frac{\sqrt{3}}{3}\\\frac{\sqrt{3}}{3}\\\frac{\sqrt{3}}{3}\end{pmatrix}, \begin{pmatrix}-\frac{\sqrt{3}}{3}\\\frac{\sqrt{3}}{3}\\\frac{\sqrt{3}}{3}\end{pmatrix}, \begin{pmatrix}-\frac{\sqrt{3}}{3}\\-\frac{\sqrt{3}}{3}\\\frac{\sqrt{3}}{3}\end{pmatrix}, \begin{pmatrix}\frac{\sqrt{3}}{3}\\-\frac{\sqrt{3}}{3}\\\frac{\sqrt{3}}{3}\end{pmatrix}.$$

Samozřejmě, že každá normála definuje přesně dvě rovnoběžné roviny, proto u 2D případu máme dohromady 8 hraničních rovin a u 3D dokonce 14 rovin. Počet normál není nikterak omezen a jejich zvýšením můžeme zkonstruovat přiléhavější obálku, avšak náročnější na výpočet.

Zbývá představit výpočet průsečíku paprsku s takto definovanou obálkou. Mějme normálu $\vec{d_k}$. Oblast, v níž leží ohraničené těleso, je určena parametry d_k^{near} a d_k^{far} (jak je vidět na obrázku 13.). Pak pro paprsek s počátkem v P_0 a směrem $\vec{P_d}$ získáme vždy dva průsečíky s rovinami normály $\vec{d_k}$. Tyto průsečíky jsou určeny dvěma parametry t_k^{near} a t_k^{far} paprsku a vypočítáme je jako

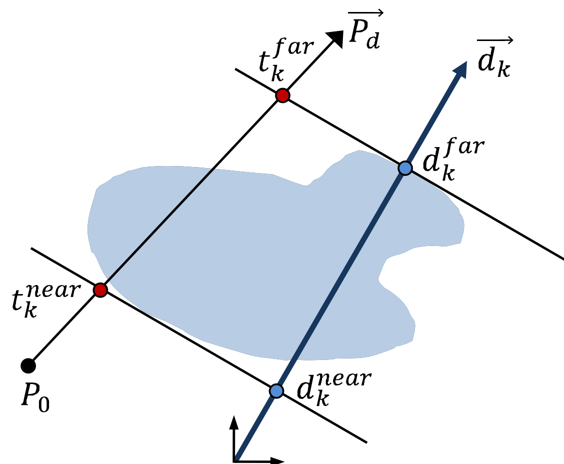
$$\begin{aligned} t_k^{near} &= \frac{d_k^{near} - \vec{d_k} * \vec{P_d}}{\vec{d_k} * \vec{P_d}}, \\ t_k^{far} &= \frac{d_k^{far} - \vec{d_k} * \vec{P_d}}{\vec{d_k} * \vec{P_d}}. \end{aligned}$$

Pozor si musíme dát na stejnou orientaci směrů. Je-li součin $\vec{d_k} * \vec{P_d}$ záporný, tak prohodíme t_k^{near} s t_k^{far}. Pro zjištění průsečíku paprsku s celou obálkou je nejprve potřeba spočítat hodnoty t_k^{near} a t_k^{far} pro každou normálu $\vec{d_k}$ zvlášť a z nich potom vybrat t^{min} a t^{max} jako

$$\begin{aligned} t^{min} &= \min_k\{t_k^{far}\}, \\ t^{max} &= \max_k\{t_k^{near}\}. \end{aligned}$$

Vyjde-li, že $t^{min} > t^{max}$, tak paprsek obálku neprotíná. Jinak průsečíky paprsku s obálkou I^{min} a I^{max} spočítáme jako

$$\begin{aligned} I^{min} &= P_0 + t^{min} \cdot \vec{P_d}, \\ I^{max} &= P_0 + t^{max} \cdot \vec{P_d}. \end{aligned}$$



Obrázek 13.: **Hraniční roviny.**

3.4.2. Hierarchie obálek (BVH)

Samotným zavedením obálek sice urychlíme algoritmus ray-tracingu, ale ještě ne razantně. Pouze by se u každého objektu nejdříve otestovalo, zda existuje průsečík paprsku s jeho obálkou, a při odpovědi ANO by se pokračovalo výpočtem průsečíku s objektem. Tato technika se již považuje za optimalizační, ale samotná se vůbec nepoužívá. Její zlepšení není dostatečné. Vždy by se muselo počítat přesně N průsečíků s obálkami a následně dalších až N průsečíků s objektem. Lze ověřit, že v některých případech je tato metoda dokonce kontraproduktivní. Konkrétně na případu scény s jedním objektem zasahujícím do celé oblasti projekční roviny. Příklad takové scény je na obrázku 14. a podrobná data lze vyčíst z tabulek 2. a 3. Z dat je patrné, že v případě scény na obrázku 14.b vychází mnohem lépe naivní algoritmus vzhledem k jedinému výpočtu průsečíku ke každému paprsku. Optimalizační metody v tomto případě totiž počítají průsečík ke každému paprsku právě dvakrát – s objektem i s obálkou. I když výpočet s obálkou je jednodušší, pořád se jedná o nezanedbatelný výpočet, a tedy optimalizace obálkou zvyšuje složitost algoritmu. Zajímavé je také pozorovat v obou tabulkách hodnoty u akcelerujících metod dělících prostor (octree a kd-tree). Vidíme, že počet testů s obálkou je také přesně roven počtu testů s objektem naivního algoritmu. Shrneme-li výše zmíněné, tak samostatné použití obálek v roli optimalizace je velmi neefektivní.

Zařadíme-li obálky do určité hierarchické struktury, výrazně zvýšíme efektivitu ray-tracingu. Takové spojení je v grafice označováno jako *BVH* (z anglického *bounding volume hierarchy*). Zdaleka nejčastější hierarchickou strukturou obálek je strom. Kromě algoritmu tvorby hierarchie se mohou stromy mezi sebou lišit počtem záznamů v uzlu nebo maximální povolenou hloubkou stromu. Kvalitní

technika optimalizace	počet výpočtů s objektem	počet výpočtů s obálkou
naivní	1 617 739	–
R-tree	1 581 501	1 617 739
Octree	1 581 501	1 617 739
Kd-tree	1 581 501	1 617 739

Tabulka 2.: **Nevýhoda samostatných obálek – k obrázku 14.a.**

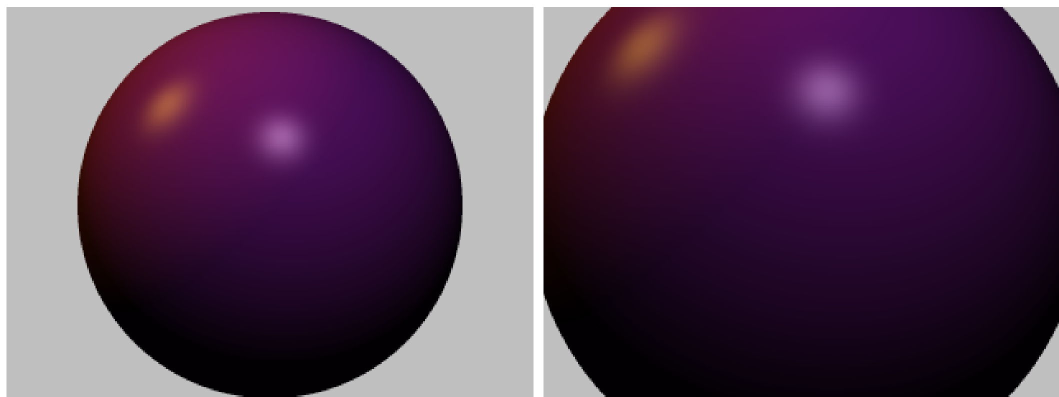
technika optimalizace	počet výpočtů s objektem	počet výpočtů s obálkou
naivní	2 174 019	–
R-tree	2 174 019	2 174 019
Octree	2 174 019	2 174 019
Kd-tree	2 174 019	2 174 019

Tabulka 3.: **Nevýhoda samostatných obálek – k obrázku 14.b.** Vidíme stejný počet výpočtů průsečíků s objektem u všech metod. Optimalizační metody vycházejí celkově mnohem hůře, jelikož navíc obsahují nezanedbatelný počet průsečíků paprsku s obálkou.

hierarchické struktury by měly mít společné některé vlastnosti.

- Každý objekt je v hierarchii umístěn právě jednou.
- Každý podstrom by měl obsahovat objekty, které nejsou příliš vzdálené. Čím více jsou uzly hlouběji v podstromu, tím jsou si i topologicky blíže.
- Obálka každého uzlu by měla být co nejmenší.
- Celkový objem všech obálek ve stromě by měl být co nejmenší.
- Důraz více kladen na kvalitně zkonstruované obálky uzlů v nízkých hloubkách stromu pro efektivní ořezávání co největších podstromů.
- Vyšší složitost konstrukce hierarchie je ospravedlnitelná při radikálním snížení celkového času ray-tracingu.

Strom lze v závislosti na vybraném algoritmu konstruovat oběma směry, typická je ovšem metoda rekurzivní konstrukce zdola nahoru. Začínáme vytvořením obálky pro každý objekt, tedy listy stromu. Následně nějakým chytrým způsobem shlukujeme obálky do otcovských uzlů, jimž vytvoříme také obálku sjednocením



(a) rozdílný počet testů paprsku s objektem a obálkou

(b) stejný počet testů paprsku s objektem i s obálkou

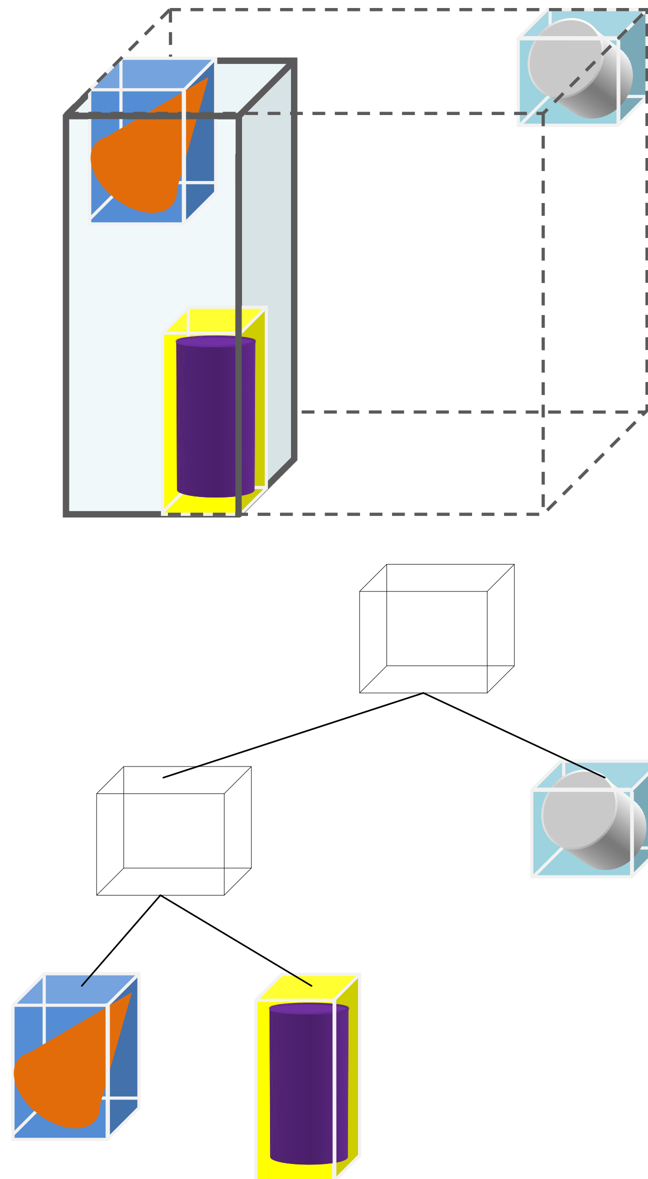
Obrázek 14.: **Obrázek k demonstraci neefektivity samostatných obálek.** Výpočet v obrázku 14.a sice obsahuje menší počet testů paprsku s objektem než naivní přístup, ale počet testů s obálkou tento rozdíl několikanásobně převyšuje. Obrázek 14.b dokonce obsahuje stejný počet testů paprsku s objektem pro optimalizaci i naivní přístup, a tedy optimalizační technika je zde definitivně pomalejší, jelikož obsahuje navíc výpočty s obálkou. Podrobnější srovnání viz tabulky 2. a 3. Balík: *Bounding volume.*

obálek z potomků. Takto postupujeme, dokud nevznikne jediný uzel stromu – kořen – reprezentující minimální obálku zahrnující celou scénu (pro ilustraci viz obrázek 15.). Při testování průniku s paprskem postupujeme od uzlu k jeho potomkům. Protne-li paprsek obálku daného uzlu, postupujeme k jeho potomkům. Jinak vynecháme celý podstrom. Dostaneme-li se procházením až k listu, tak teprve potom budeme počítat průsečík paprsku s listovým objektem. Je zřejmá teoretická logaritmická složitost oproti lineární složitosti naivního algoritmu při dobře postaveném stromě. Samozřejmě není možné zaručit logaritmickou složitost u každé scény. Podrobněji popíšeme jeden z takto pracujících algoritmů – konkrétně *STR packing* algoritmus pro R-tree (viz kapitolu 3.4.3.), avšak tento algoritmus je pouze jedním z mnoha způsobů jak přistupovat k principu hierarchie obálek (například viz [10]).

BVH jsou schopné přizpůsobit se libovolnému rozdělení objektů ve scéně (na rozdíl např. od mřížky) a dají se poměrně snadno vytvořit (jako např. mřížka).

3.4.3. R-tree

S rozvojem algoritmů pracujících s geometrickými daty, jako jsou body a hyper-povrchy z vyšších dimenzí, se vyvinuly i struktury k jejich efektivnímu uložení a vyhledávání. Mezi tyto struktury patří poměrně nové R-tree. Obecně jsou to struktury pro práci s multi-dimenzionálními daty, obrázky, geografickými



Obrázek 15.: **BVH.** Hierarchie obálek. V listech jsou uloženy obálky všech objektů scény. Vnitřní uzel obsahuje obálku ke všem jeho potomkům.

daty a rozsáhlými prostorovými databázemi. Označení R-tree (podobně jako B-tree) v sobě zahrnuje celou rodinu struktur a jejich operací. Mezi nejznámější jsou např. R^+-tree, R^*-tree, Sphere-tree, Hilbert Packed R-tree atd. Stromy mohou být buď dynamické, nebo statické. Dynamické označení je pro ty, které se mohou neustále měnit. U statických je výhoda, že všechna ukládaná data jsou známa předem a můžeme vytvořit optimálně vyvážený strom.

R-tree jsou hierarchické datové struktury na bázi B^+-tree. Slouží k organizaci množiny d-dimenzionálních geometrických objektů. Ukládaná data jsou reprezentována tzv. *MBR* – minimální d-dimenzionální obálkou[1] (v angličtině *minimum bounding d-dimensional rectangle*). Každému uzlu ve stromě je přiřazena MBR, která obaluje všechny jeho potomky resp. jejich MBR. Listy stromu obsahují ukazatele na data a zároveň rovněž jejich MBR. Obálky se mohou překrývat, avšak každá z nich může patřit pouze jednomu uzlu. Snaha je vytvořit vždy co nejmenší MBR vnitřních uzlů (u listů je MBR již dána jejich daty). O R-tree řekneme, že je řádu (m, c), když

- počet záznamů každého uzlu (kromě kořene) je nejméně m a nejvíce c, přičemž $m \leq \frac{c}{2}$,
- minimální počet záznamů kořene je 2 (je-li kořen jediný uzel stromu, tak i méně),
- všechny listy leží na stejné úrovni.

R-tree jsou tedy z definice vyvážené. Obsahuje-li R-tree množinu dat velikosti N, tak jeho maximální výška bude

$$h_{max} = \lceil \log_m N \rceil - 1.$$

Základní operací na R-tree je *RangeSearch*. Pro zadanou MBR *Qmbr* najde všechna data, jejichž MBR protíná *Qmbr*. Nebudeme zde podrobně popisovat algoritmy na R-tree, pro detaily viz [9]. Zaměříme se však na stromy pracující se statickými daty, u nichž operace vkládání a mazání nejsou prováděny (nebo jen zřídka). Máme-li všechna data již před vytvořením stromu, můžeme toho využít a efektivně navrhnout strom pouze pro ně, čímž zvýšíme výkon operace vyhledávání (vytvoříme optimální počet uzlů, optimální výšku stromu, minimální překrývání mezi MBR). Metody, které se zaměřují jen na statická data, se nazývají *pakovací* (z angl. *packing* nebo *bulk loading*). Tímto se dostáváme k BVH, jelikož R-tree se nabízí jako vhodná hierarchická struktura u akcelerujících metod dělení objektů a pakovací algoritmy k její konstrukci.

V dalším budeme předpokládat, že vstupní množina dat je velikosti N a uzel má kapacitu c. MBR budeme označovat pro lepší skloňování výhradně jako

[1] V tomto okamžiku je jistě již zřejmá analogie mezi R-tree a BVH, speciálně mezi MBR a AABB. Konkrétně, R-tree je druh obálkové hierarchie a MBR je jiné označení pro AABB v trojrozměrném prostoru.

obálku. První pakovací algoritmus z roku 1985 je velmi jednoduchý – viz algoritmus 3.3. Tento algoritmus však nezpracovává vstupní data příliš chytře – třídí

Algoritmus 3.3. Pakovací algoritmus

Vstup: množina dat velikosti N, maximální kapacita uzlu c
Výstup: kořen stromu
Ze vstupních dat vytvoř samostatné uzly stromu – N listů.

1. Seřaď obálky podle jedné ze souřadnic jejich středu.
2. Obálky rozděl do $\lceil \frac{N}{c} \rceil$ skupin – každá bude obsahovat c obálek (poslední skupina jich může obsahovat méně).
3. Ke každé skupině vypočítej společnou obálku, jejíž data budou původní obálky.
4. Z každé nové obálky vytvoř uzel stromu, jehož potomci budou uzly původních obálek.
5. Vznikly-li více jak 2 uzly, opakuj kroky 1 – 4 na nově vzniklé uzly pro následující souřadnici. Souřadnice střídej cyklicky.

---

je cyklicky podle všech souřadnic, aniž by věděl, která je pro daná data výhodnější. Strom tedy nemusí být optimálně přizpůsoben vstupním datům. Efektivnější rozdělení obálek v prostoru poskytuje algoritmus *Hilbertovo pakování* využívající Hilbertovu křivku. Každé listové obálce je určena Hilbertova hodnota jejího středu a následně jsou všechny obálky podlé této hodnoty setříděny. Strom se pak konstruuje zdola nahoru sjednocením obálek z nižší úrovně. Kompletní algoritmus viz [9].

Experimentálně je ověřeno, že následující pakovací algoritmus poskytuje lepší výsledky, než předchozí dva algoritmy (Hilbertova metoda je jen ve výjimečných případech o trochu lepší). Jedná se o *STR* (*Sort-Tile-Recursive*) algoritmus. Je navržen pro libovolně velké dimenze vstupních dat a libovolnou kapacitou uzlů. STR postupuje rekurzivně po jednotlivých dimenzích. Nejdříve vstupní hyperkrychle (dimenze d a velikosti N) setřídí podle první dimenze jejich středu. Data pak rozdělí do tzv. *řezů* velikosti $\left\lceil \lceil \frac{N}{c} \rceil^{\frac{1}{d}} \right\rceil$, kde každý řez obsahuje $\left\lceil \lceil \frac{N}{c} \rceil^{\frac{d-1}{d}} \right\rceil \cdot c$ hyper-krychlí. Na každý nový řez je algoritmus zavolán rekurzivně s dimenzí velikosti $d-1$ (data budou velikosti právě $N = \left\lceil \lceil \frac{N}{c} \rceil^{\frac{d-1}{d}} \right\rceil \cdot c$). Nerekurzivní algoritmus pro dimenzi 3 viz algoritmus 3.5..

Konstrukce stromu pak probíhá zdola nahoru. Na vstupu dostaneme N obálek, které nejprve setřídíme algoritmem STR a uděláme z nich uzly stromu (nových uzlů bude vždy $\lceil \frac{N}{c} \rceil$). Pro každých c uzlů vytvoříme jejich rodičovský uzel a spočítáme jeho obálku z potomků. Vytvoříme-li vždy více než jeden nový uzel,

rekurzivně opakujeme s obálkami rodičovských uzlů. Kompletní postup viz algoritmy 3.4. a 3.5.

Algoritmus 3.4. Konstrukce stromu využitím STR packing algoritmu

Vstup: obálky MBR, maximální kapacita uzlu c, velikost dimenze d
Výstup: kořen stromu
while card$(MBR) > c$ **do**
1. Setřiď obálky STR algoritmem: $MBR_STR = STR(MBR, c, d)$.
2. Pro každých c obálek z MBR_STR vytvoř jejich rodičovský uzel, jehož obálka bude sjednocením obálek potomků.
3. MBR = obálky všech rodičovských uzlů.

end while

Algoritmus 3.5. STR packing algoritmus (nerekurzivní pro $d = 3$)

Vstup: obálky MBR (velikosti N), kapacita uzlu c, dimenze d
Výstup: setříděné skupiny obálek
$STR(MBR, c, d)$

1. Setřiď vstupní obálky podle první souřadnice jejich středu.
2. Obálky rozděl do $\left\lceil \left\lceil \frac{N}{c} \right\rceil^{\frac{1}{d}} \right\rceil$ skupin; každá bude obsahovat $\left\lceil \left\lceil \frac{N}{c} \right\rceil^{\frac{d-1}{d}} \right\rceil \cdot c$ obálek (poslední skupina jich může obsahovat méně).
3. Každou skupinu setřiď podle druhé souřadnice středů jejich obálek.
4. Rozděl každou skupinu do $\left\lceil \left\lceil \frac{\left\lceil \left\lceil \frac{N}{c} \right\rceil^{\frac{d-1}{d}} \right\rceil \cdot c}{c} \right\rceil^{\frac{1}{d-1}} \right\rceil$ dalších skupin; každá bude obsahovat $\left\lceil \left\lceil \frac{N}{c} \right\rceil^{\frac{1}{d}} \right\rceil \cdot c$ prvků.
5. Každou skupinu setřiď podle třetí souřadnice středů jejich obálek.

V našem případě budeme pracovat s dimenzí velikosti $d = 3$. Ideální kapacita uzlu se ukázala $c = 4$. Nyní dokážeme, že takto zvolená konstrukce skupin obálek a jejich velikosti jsou korektní. Na první pohled se totiž může zdát, že prací s odmocninami nemůžou vyjít správné počty uzlů. Nechť označuje S_1 počet prvních skupin obálek seřazených podle první souřadnice a nechť $Size_1$ označuje

počet obálek v každé této skupině:

$$\begin{aligned} S_1 &= \left\lceil \left\lceil \frac{N}{c} \right\rceil^{\frac{1}{3}} \right\rceil, \\ Size_1 &= \left\lceil \left\lceil \frac{N}{c} \right\rceil^{\frac{2}{3}} \right\rceil \cdot c. \end{aligned}$$

Pak musí platit, že $S_1 \cdot Size_1 = N$:

$$S_1 \cdot Size_1 = \left\lceil \left\lceil \frac{N}{c} \right\rceil^{\frac{1}{3}} \right\rceil \cdot \left(\left\lceil \left\lceil \frac{N}{c} \right\rceil^{\frac{2}{3}} \right\rceil \cdot c \right) = \frac{N}{c} \cdot c = N.$$

Nyní provedeme druhé kolo výpočtu pro $N = \left\lceil \left\lceil \frac{N}{c} \right\rceil^{\frac{2}{3}} \right\rceil \cdot c$ a získáme S_2 skupin, každou po $Size_2$ obálkách:

$$\begin{aligned} S_2 &= \left\lceil \left\lceil \frac{\left\lceil \left\lceil \frac{N}{c} \right\rceil^{\frac{2}{3}} \right\rceil \cdot c}{c} \right\rceil^{\frac{1}{2}} \right\rceil = \left\lceil \left\lceil \frac{N}{c} \right\rceil^{\frac{2}{3}} \right\rceil^{\frac{1}{2}} = \left\lceil \left\lceil \frac{N}{c} \right\rceil^{\frac{1}{3}} \right\rceil, \\ Size_2 &= \left\lceil \left\lceil \frac{\left\lceil \left\lceil \frac{N}{c} \right\rceil^{\frac{2}{3}} \right\rceil \cdot c}{c} \right\rceil^{\frac{1}{2}} \right\rceil \cdot c = \left\lceil \left\lceil \left\lceil \frac{N}{c} \right\rceil^{\frac{2}{3}} \right\rceil^{\frac{1}{2}} \right\rceil \cdot c = \left\lceil \left\lceil \frac{N}{c} \right\rceil^{\frac{1}{3}} \right\rceil \cdot c. \end{aligned}$$

Teď musí platit, že $S_2 \cdot Size_2 = Size_1$:

$$S_2 \cdot Size_2 = \left\lceil \left\lceil \frac{N}{c} \right\rceil^{\frac{1}{3}} \right\rceil \cdot \left\lceil \left\lceil \frac{N}{c} \right\rceil^{\frac{1}{3}} \right\rceil \cdot c = \left\lceil \left\lceil \frac{N}{c} \right\rceil^{\frac{2}{3}} \right\rceil \cdot c = Size_1.$$

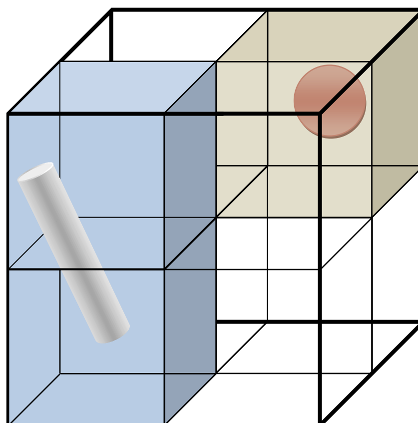
Víme-li, že

$$Size_1 = \frac{N}{S_1},$$

tak dohromady získáme:

$$\begin{aligned} S_2 \cdot Size_2 &= \frac{N}{S_1}, \\ S_1 \cdot S_2 \cdot Size_2 &= N, \end{aligned}$$

což jsme chtěli ověřit. Abychom tyto hodnoty vysvětlili – jedním STR pakováním jsme vlastně vytvořili trojrozměrné pole o rozměrech $S_1 \times S_2 \times Size_2$, jehož postupným procházením vytváříme nové obálky. V otcovském uzlu pak vznikne co nejmenší obálka, jelikož tyto obálky jsou si ve struktuře velmi blízko (leží hned vedle sebe – spíše za sebou).



Obrázek 16.: **Mřížka.** Uniformní mřížka pro scénu složenou ze dvou objektů. Zvýrazněny jsou jediné tři voxely obsahující objekt.

Popsali jsme pakovací algoritmy pracující zdola nahoru. Nejdříve vytvoří listy stromu náležící vstupním datům a konstruují strom až ke kořenu. Existují však i pakovací algoritmy pracující shora dolů. Konkrétně *VAMSplit R-tree*, což je variace kd-tree, nebo *Top-Down Greedy Split*. Podrobnější informace lze najít v [9].

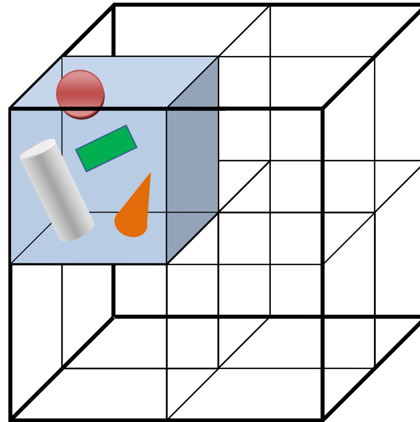
3.4.4. Mřížka

Mřížka je akcelerující struktura založená na principu rozdělení prostoru. Představme si celý prostor scény jako jedno velké rovnostranné AABB, tedy prostorovou krychli (v počítačové grafice označovanou jako *voxel*). Když hlavní krychli uniformně rozdělíme na menší voxely, obdržíme rozdělení prostoru nazvané mřížka (grid). Každý voxel pak obsahuje odkazy na všechny objekty, které do něj zasahují. Pro ukázku mřížky vytvořené ke scéně se dvěma objekty viz obrázek 16.

Hledáme-li průsečík paprsku s objekty, procházíme postupně všechny voxely, které paprsek protíná. Pořadí voxelů je určeno vzestupně podle vzdálenosti od počátku paprsku. U každého takového voxelu pak hledáme průsečík paprsku s jeho objekty. Tento postup má výhodu, že při nalezení prvního průsečíku s objektem můžeme přestat prohledávat následující voxely (zbývající objekty ve voxelu ale otestovat ještě musíme).

K zjištění voxelů protnutých paprskem se efektivně využívá klasický Bresenhamův algoritmus pro kreslení úsečky. V něm se inkrementálně hledají kreslené pixely jeden po druhém použitím jednoduchých operací součtu a porovnání. Zde však máme voxely a chceme je zjistit všechny podél paprsku (na rozdíl od Bresenhama, který všechny pixely procházející úsečkou nevykresluje).

Struktura mřížky je jednoduše implementovatelná, dá se rychle vytvořit a nezabírá moc paměti. Ale má jednu podstatnou nevýhodu. Vůbec se totiž nepřizpů-



Obrázek 17.: **Nevýhoda mřížky.** Mřížka se nepřizpůsobuje k rozložení objektů ve scéně.

sobuje rozložení objektů ve scéně (jako příklad viz obrázek 17.). Máme-li scénu s nerovnoměrným rozložením objektů (typický případ), tak spousta objektů padne do několika málo voxelů a výkon mřížky značně klesne. Vždy u takového voxelu budeme totiž testovat s paprskem i většinu objektů ve scéně[1]. Samozřejmě, že vždy můžeme vytvořit i menší voxely, aby v každém z nich bylo co nejméně objektů, nicméně ne vždy to pomůže. Je-li mřížka příliš hustá (obsahuje spoustu malých voxelů), mnoho času trvá výpočet voxelů incidujících s paprskem. Když je mřížka příliš volná, mnoho objektů padne do každého voxelu. Vzhledem k těmto nepříjemným vlastnostem se mřížka jako akcelerující struktura téměř nevyužívá.

Jeden z hlavních problémů mřížky je také mnohonásobné odkazování na jeden objekt z několika oblastí. Technika, která se tímto problémem vypořádává, se nazývá *mailboxing* (nebo také *timestamping*) a pracuje na jednoduchém principu, při kterém se každý objekt během prvního výpočtu označí. Před každým výpočtem se pak ověří, zda byl objekt již označen a v tom případě se celý výpočet přeskočí. I když tímto dojde k zamezení spousty výpočtů, stále máme mnoho duplikovaných objektů, což stojí čas i paměť.

[1] Takovému problému se říká „jedna konvice s čajem na stadionu“.

3.4.5. Octree

K technikám dělícím prostor můžeme přistupovat z několika směrů. Buď budeme prostor dělit pravidelně (případ mřížky a octree) nebo prostor budeme dělit podle rozložení objektů v něm (kd-tree a BSP-tree). Nejprve si představíme jednodušší způsob a jeho principy na struktuře octree. Název octree je odvozen od počtu částí, na které je trojrozměrný prostor rozdělen – tedy 8. Můžeme se setkat s pojmem *quadtree*, jenž dělí 2D prostor na 4 části. Způsob dělení prostoru je jednoduchý, jelikož hraniční roviny jsou opět jako u mřížky kolmé k souřadnicovým osám. Máme tedy oblast tvaru AABB, jež se vždy dělí právě na 8 stejně velkých AABB oblastí. Octree můžeme abstraktně definovat jako rekurzivní mřížku. Zjednodušeně řečeno, v n-rozměrném prostoru rozdělíme interval na každé ose na dvě poloviny. Konkrétně v quadtree se jedná o 2×2 nových intervalů a u octree dělíme ještě jeden interval navíc, takže vznikne celkem $2 \times 2 \times 2$ nových intervalů. V každém kroku vznikne vždy 2^n nových oblastí. Jelikož se jedná o strom, tak každá AABB oblast reprezentuje jeden uzel stromu a po rozdělení vznikne 2^n jeho potomků.

Mějme opět libovolnou netriviální scénu a k ní vytvořenou celkovou AABB (tedy sjednocení obálek všech objektů). Stejně jako u mřížky, každá AABB bude odkazovat na objekty, které jsou v ní obsaženy. Rozdíl je ale v tom, že u octree bude objekt obsažen právě v jednom AABB ze všech, které jej překrývají. V tomto kritériu zařazení objektů do oblastí se často vyskytuje rozpor. Některé zdroje uvádí (např. [10]), že oblasti mají obsahovat všechny objekty, které je překrývají (neefektivní, musí se řešit *mailboxingem*). Jiné tvrdí ([7]), že každý objekt má být obsažen právě v jedné oblasti. Pravdou je, že možných způsobů je prostě více a nelze jednoduše rozhodnout, který je lepší. V [1] jsou uvedeny další techniky. Objekt například nemusí být uložen v listu, ale v posledním uzlu, který jej plně obsahuje (neefektivní, když je objekt uprostřed oblasti). Objekt může být také rozpůlen na dva disjunktní objekty (neefektivní, protože vzniká více objektů). Klasický octree můžeme modifikovat na tzv. *uvolněny octree*, jehož oblasti jsou lehce zvětšeny a vzájemně se překrývají. Objekty pak můžeme přiřadit pouze jedné oblasti. Záleží však na konkrétním způsobu konstrukce a implementace. Například v [10] pracují s body, které jsou rovnou středy obálek, a tedy není jasné, do jakých všech oblastí objekt zasahuje. Navíc tímto způsobem může bod zasahovat do více oblastí jen v případě, pokud leží na jejich hranici. Zato v [7] pracují s celým objektem, respektive se všemi hraničními body jeho obálky. Je otázka, který přístup je lepší. V programu RayTracer byla použita metoda středů obálek a jedinečnost objektů v listech (tedy kombinace obou principů).

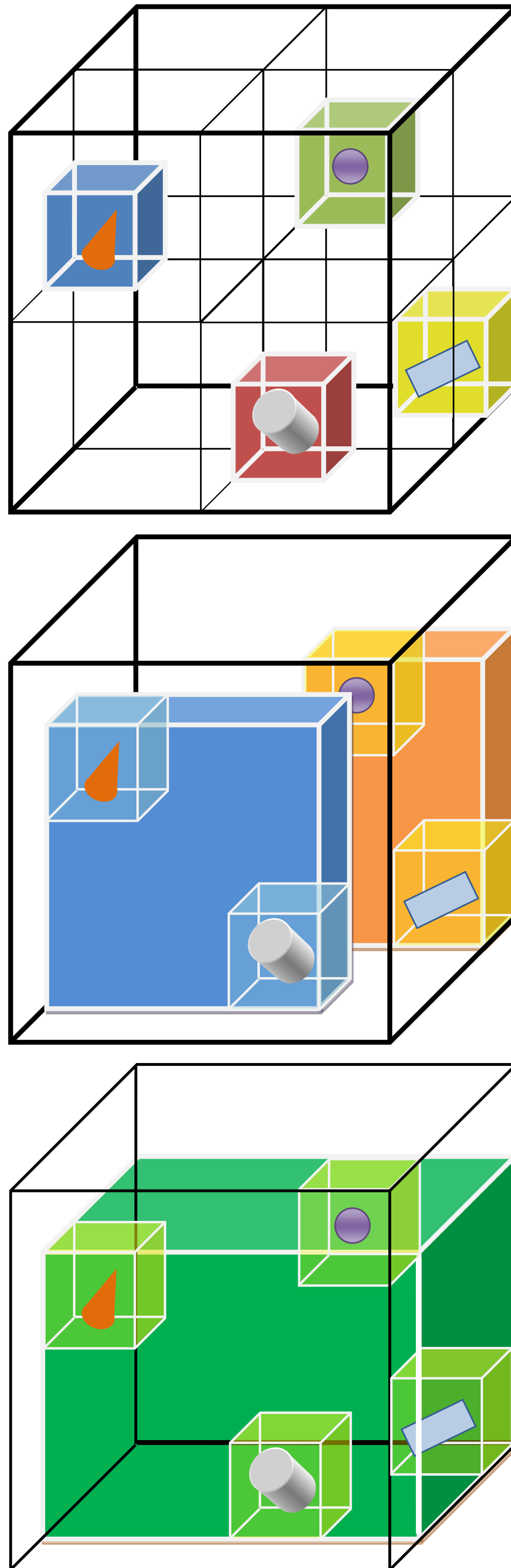
Obsahuje-li AABB více jak povolenou kapacitu objektů, rozdělíme oblast a přerozdělíme objekty do nově vytvořených oblastí. Kritérií k zastavení rekurze však bývá více: maximální hloubka stromu, minimální délka intervalu oblasti, minimální počet objektů v oblasti. Tento základní algoritmus je velmi jednoduchý, ale dá se na něm ještě mnoho vylepšit. Z konstrukce je zřejmé, že algoritmus

pracuje shora dolů a kompletně dělí prostor (ve smyslu neexistence prázdného místa – neplést si s oblastí, která neobsahuje žádný objekt). Můžeme tedy strom po jeho vytvoření upravit procházením ještě zdola nahoru. V listech vytvoříme minimální obálku objektů a v rodičovských uzlech sjednotíme obálky potomků. Takto budeme postupovat až ke kořenu stromu (pro představu viz obrázek 18.). Navíc oblastem neobsahujícím žádné potomky přiřadíme prázdný odkaz. Výrazně tím zvýšíme výkon při hledání průsečíku paprsku s objektem, protože místo testování všech oblastí původního octree budeme testovat jen obálky. Ve skutečnosti octree ani akcelerující strukturou není, jelikož nepřináší téměř žádný benefit vzhledem k úplnému dělení prostoru. Octree je pouze prostředek k vytvoření hierarchie obálek.

Další vylepšení BVH vytvořené pomocí octree se dá provést při procházení stromu. Protíná-li paprsek obálku nějakého uzlu a zároveň protíná alespoň dvě obálky v jeho potomcích, tak bychom měli pokračovat v testování takového uzlu, jehož obálka má průsečík s paprskem nejblíže. Máme tak totiž větší šanci, že v takovém uzlu bude ležet i objekt blíže paprsku. Kay a Kajiya v [7] navrhují při procházení stromu řadit obálky vzestupně podle vzdálenosti jejich průsečíku od počátku paprsku. Postupujeme pak vždy do uzlu s nejbližší obálkou a při průsečíku jeho některých potomků, vložíme také jejich obálky do seznamu na začátek, jelikož jejich průsečík je nutně blíže počátku paprsku než ostatní obálky v seznamu. Seznam obálek bude tedy prioritní fronta. Nejvyšší prioritu zde bude mít právě obálka s nejmenší vzdáleností od počátku paprsku. Při procházení vyjmeme první obálku z fronty. Náleží-li listu stromu, tak ze všech jeho objektů zjistíme nejbližší průsečík s paprskem. Našel-li se průsečík a je-li jeho vzdálenost od počátku paprsku menší, než vzdálenost první obálky ve frontě od počátku paprsku, můžeme ukončit prohledávání (není totiž možné, aby ve frontě existovala obálka s objekty bližšími k počátku paprsku než nalezený průsečík). Jinak pokračujeme dál v prohledávání, dokud nenajdeme první bližší průsečík nebo nevyprázdníme frontu.

3.4.6. Kd-tree

Kd-tree, stejně jako octree, je metoda dělící prostor. Na rozdíl od octree je kd-tree binární a místo pravidelného rozdělení oblasti na 8 částí ji dělíme pouze na části dvě. Zato se však snažíme přizpůsobit strukturu k rozdělení objektů v prostoru, aby nevznikaly zbytečně prázdné oblasti. Představme si scénu, kde 99% objektů leží v jedné oblasti, která je až hluboko v octree. Pak tato struktura je velmi neefektivní, jelikož strom bude nevyvážený. Obecně octree nebývají dobře vyvážené. Kd-tree tento problém opravuje, jelikož se pokouší dělit prostor nestejnorodě. Dělíme-li ale oblast na dvě menší, je zde otázka, podle jakých os a ve kterém místě dělení provést. Existuje několik metod, jak dělení provádět. Například cyklicky střídat osy a dělit je v polovině intervalu (tím bychom dostali přesně octree, tedy jeho binární verzi). Dobrý kd-tree dělí prostor na poloviny, ale



Obrázek 18.: **Tvorba minimální bounding volume pro octree.**

ne geometricky, nýbrž numericky (na dvě stejně mohutné oblasti). Jeden z nejlepších způsobů dělení prostoru kd-tree je tzv. *řez mediánem*. Rovnou ale představíme jeho optimalizovanou verzi. Máme-li dělenou oblast definovanou třemi osami a jejich intervaly, vybereme jako dělící tu osu, jejíž interval je nejdelší. Podle této osy setřídíme všechny body v oblasti a vybereme prostřední z nich. Jeho souřadnice bude určovat dělící souřadnici na vybrané ose. Nakonec vytvoříme dvě nové oblasti a každé přiřadíme příslušnou polovinu bodů. Při tom konstruujeme strom opět shora dolů. Ukončovací kritérium bude maximální hloubka stromu, minimální počet objektů v oblasti, nebo minimální délka intervalu.

Stejně jako u octree, po vytvoření stromu je dobré provést optimalizaci a rekonstruovat strom zdola nahoru, kdy každému uzlu přiřadíme obálku sjednocenou z obálek jeho potomků. Ještě připomeneme, že vstupní oblast je AABB obálka sjednocená z obálek objektů scény a body jsou většinou středy obálek.

3.4.7. BSP-tree

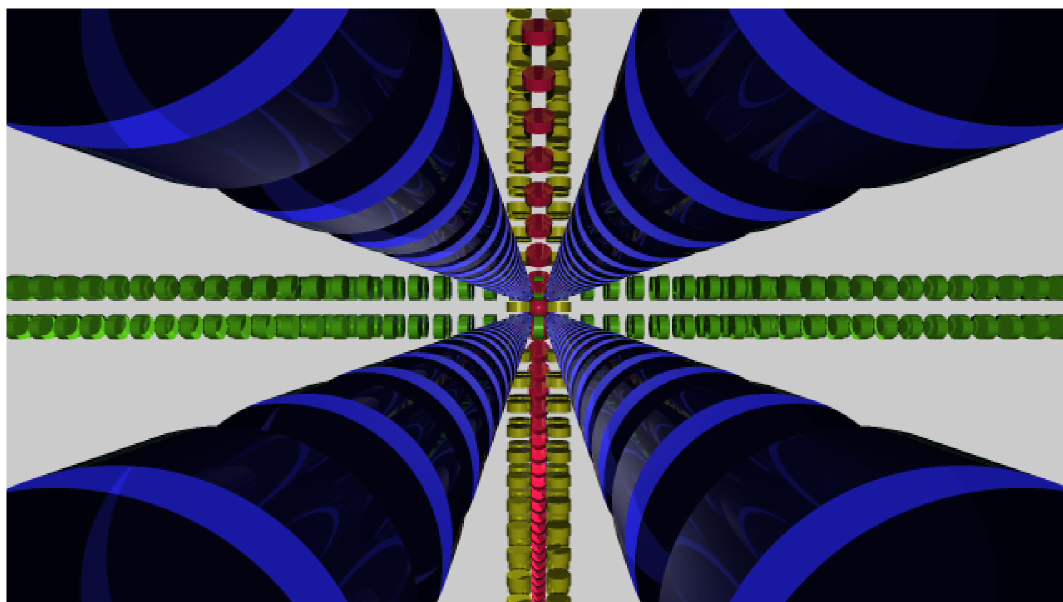
Hierarchie obálek BSP-tree se v ray-tracingu příliš nepoužívá, ale zmíníme, že BSP-tree je obecnější verze kd-tree. Jeho dělící roviny jsou totiž obecného typu a proto jednak náročnější na paměť a jednak je pro ně náročnější výpočet průsečíku paprskem než pro AABB. Ale dříve se používaly např. v hrách jako byl *Quake* nebo *Doom* při řešení viditelnosti nebo detekci kolizí. BSP-tree byly typické pro vnitřní scény, zato kd-tree a octree se používaly pro venkovní scény.

3.4.8. Srovnání metod

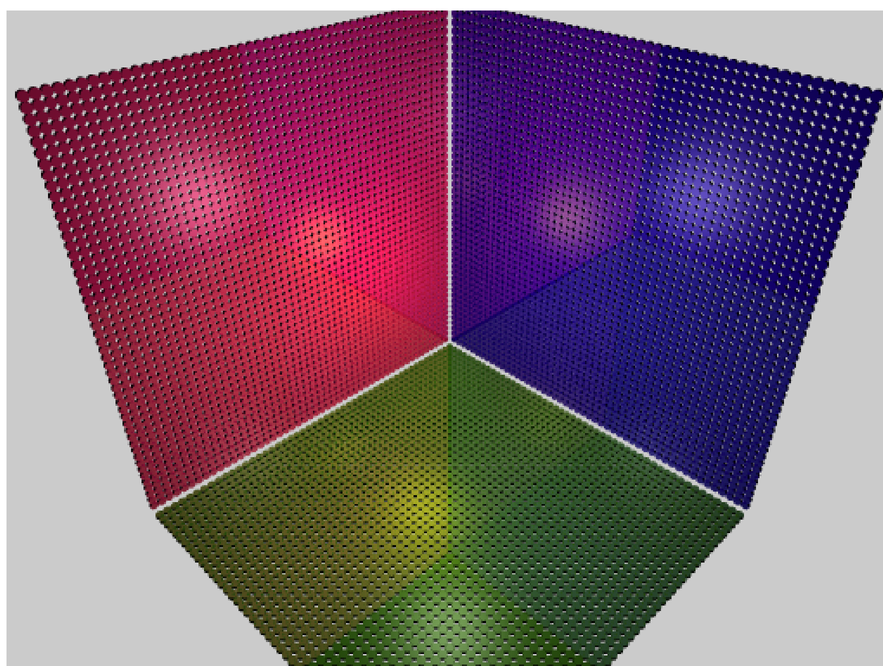
Popsali jsme populární techniky hierarchických akcelerujících struktur ray-tracingu a zbývá určit, která z nich je nejlepší. Viděli jsme i případ, kdy přinesly spíše zhoršení (viz obrázek 17.). Pravdou je, že jsou techniky dělení prostoru i objektů v průměru srovnatelné. Ani jedna technika není totiž nejlepší na všechny typy scén. Každá se hodí na jinou scénu podle zvolené geometrie, přístupu ke strukturám a každé z nich trvá jiný čas tvorby hierarchie. Důležité je především zlepšení od naivního principu, kdy jsme vždy viděli až několika-řádové zlepšení. Většinou se ale udává, že nejlepší výsledky poskytuje kd-tree, vzhledem k jeho nejlepšímu přizpůsobení oblasti k rozložení objektů ve scéně. Shodou okolností podobné výsledky vychází i na všech našich datech, ačkoli rozdíly mezi kd-tree a R-tree jsou často zanedbatelné. R-tree sice také přizpůsobuje obálky objektům, a i když má někdy méně výpočtů průsečíků paprsku s objekty, vždy obsahuje více výpočtů s obálkami. Výkon octree se přitom pohybuje na srovnatelné úrovni obou metod.

3.5. Optimalizace vržených stínů

Víme, že v každém bodě průsečíku paprsku s objektem je vždy nutné zjistit,



Obrázek 19.: **Scéna k porovnání metod 1.** Scéna obsahuje 1 313 válců a dvě bodová světla. Rozlišení obrázku je 640 × 360 s hloubkou rekurze 1 a antialiasingem. Srovnávací data viz tabulka 4. Balík: *cylAxes*.



Obrázek 20.: **Scéna k porovnání metod 2.** Scéna obsahuje 7 500 válců a dvě bodová světla. Rozlišení obrázku je 512 × 384 s hloubkou rekurze 1 a antialiasingem. Srovnávací data viz tabulka 5. Balík: *cylWalls*.

technika optimalizace	počet výpočtů s objektem	počet výpočtů s obálkou
naivní	$24\,225 \cdot 10^6$	–
R-tree	$95 \cdot 10^6$	$1\,033 \cdot 10^6$
Octree	$94 \cdot 10^6$	$466 \cdot 10^6$
Kd-tree	$92 \cdot 10^6$	$608 \cdot 10^6$

Tabulka 4.: **Data ke scéně z obrázku 19.** Počet průsečíků s objekty je u všech tří optimalizačních metod přibližně srovnatelný. Nejhůře ale vychází R-tree jak pro testy s objekty, tak pro testy s obálkami. Metody octree a kd-tree jsou na stejné úrovni, i když octree má méně testů s obálkami, ale více testů s objekty.

technika optimalizace	počet výpočtů s objektem	počet výpočtů s obálkou
naivní	$67\,086 \cdot 10^6$	–
R-tree	$21 \cdot 10^6$	$308 \cdot 10^6$
Octree	$21 \cdot 10^6$	$144 \cdot 10^6$
Kd-tree	$21 \cdot 10^6$	$145 \cdot 10^6$

Tabulka 5.: **Data ke scéně z obrázku 20.** Data byla pořízena pro obrázek s rozlišením 320×240. Doba výpočtu naivní technikou byla téměř 10 hodin, oproti necelým 7 minutám u optimalizačních metod.

zda a jaká světla jej osvětlují. To provedeme vysláním tzv. *stínového paprsku* z bodu průniku P směrem ke světlu. Tím pádem musíme opět zjistit všechny objekty, které tento paprsek protíná. V naivním algoritmu se znovu procházel každý objekt a nakonec z bodů průniku se vybral ten nejbližší Q k počátku P stínového paprsku. Jestli vzdálenost mezi těmito body byla menší než vzdálenost bodu P od světla, tak bod P byl před tímto světlem zastíněn (detaily opět viz [2]). Problém zde spočívá ve fázi testování průniku těles se stínovým paprskem. Přece nemusíme vždy najít všechny jeho body průniku a už vůbec ne ten nejbližší z nich. Nově navržená optimalizace je velmi jednoduchá a docela účinná. Spočívá v nalezení pouze prvního stínícího objektu a ořezání zbývajících výpočtů. Před vysláním stínícího paprsku stačí spočítat vzdálenost bodu P ke světlu a přidat tuto informaci k nosnému paprsku. Po každém novém průniku se zároveň spočítá vzdálenost k počátku paprsku a je-li menší, než vzdálenost ke světlu, končíme výpočet a objekt je zastíněn.

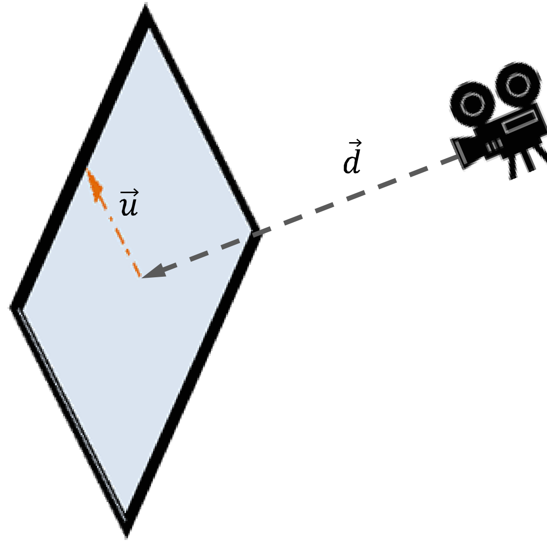
Touto metodou si výrazně urychlíme výpočty ve scénách s měkkými stíny (hlavně při multi-pass výpočtech). Vrátíme-li se k obrázku 10. (scéna obsahuje dvě

bodová světla) a použijeme-li právě zmíněnou optimalizaci[1], dostaneme zajímavá data znázorněná v tabulce 6. U optimalizační struktury je zlepšení „pouze“ asi 14%, zato u naivního algoritmu se zlepšení blíží dokonce 28%.

technika optimalizace	**počet výpočtů s objektem**	**počet výpočtů s obálkou**
naive shadow-naive	$3\,169,1 \cdot 10^6$	–
naive shadow-opt	$2\,293,9 \cdot 10^6$	–
Kd-tree shadow-naive	$0,7 \cdot 10^6$	61 904 812
Kd-tree shadow-opt	$0,6 \cdot 10^6$	61 904 812

Tabulka 6.: **Porovnání optimalizací vržených stínů.** Vidíme nemalé zlepšení jak u optimalizace kd-tree, tak u naivního principu.

[1] V programu RayTracer nelze přepínat volbu mezi optimalizací vržených stínů a bez nich. Tato optimalizace je v programu implementována implicitně, jelikož by různě volitelných optimalizačních kombinací bylo v programu příliš mnoho.



Obrázek 21.: **Pohled kamery.**

4. Editor

Tvorba trojrozměrného editoru se skládá ze spousty dílčích operací. Některé jsme již ovšem uvedli vzhledem k jejich univerzálnímu uplatnění napříč různými oblastmi počítačové grafiky. Jednalo se např. o homogenní souřadnice (kapitola 2.1.1.), afinní transformace (kapitola 2.1.2.), faktorizace matice na Eulerovy úhly (kapitola 2.2.), převod vektorů na kvaternion (kapitola 2.3.). V editoru programu RayTracer bylo použito několik dalších metod, z nichž vybereme ty nejzajímavější. První z nich je nastavení orientace rotační matice editoru podle směru pohledu kamery (viz kapitolu 4.1.). Jako druhou operaci představíme výpočet bodů obecné elipsy (viz kapitolu 4.2.).

4.1. Pohled kamery

Trojrozměrný editor v programu RayTracer umožňuje uživateli vytvářet scény pro ray-tracing. Důležitým požadavkem na každý editor bývá možnost zobrazení scény uživateli z pohledu kamery. Již v předešlé práci [2] jsme vysvětliti funkci kamery a její definici. Kamera je zadaná polohou a dvěma směry. První vektor $\vec{d}$ ukazuje směrem vpřed kamery, druhý vektor $\vec{u}$ směřuje nahoru (jak je vidět na obrázku 21.). Oba vektory jsou na sebe kolmé. Cílem je, aby rotační matice editoru byla orientována přesně podle těchto vektorů a pozorovatel byl umístěn v poloze kamery. Existuje více způsobů jak tento problém řešit – například použitím kvaternionů. V kapitole 2.3. jsme popsali konstrukci kvaternionu převádějící jeden vektor na druhý. Přesně takový výpočet zde použijeme a rovnou

dvakrát. Musíme však vědět, jaký způsob orientace souřadnicových os používá editor. Nejdříve vytvoříme první kvaternion q_1 převádějící vektor $\vec{d}$ na vektor osy z editoru a z něj vypočítáme rotační matici R_1 (viz rovnici (2)). Dále transformujeme souřadnice vektoru $\vec{u}$ maticí R_1^T:

$$\vec{u'} = R_1^T \cdot \vec{u}.$$

Následně z vektoru $\vec{u'}$ a z vektoru opačného k vektoru osy y vytvoříme kvaternion q_2 a opět z něj vyjádříme rotační matici R_2. Výsledná rotační matice R editoru pak bude tvaru

$$R = R_1 \cdot R_2.$$

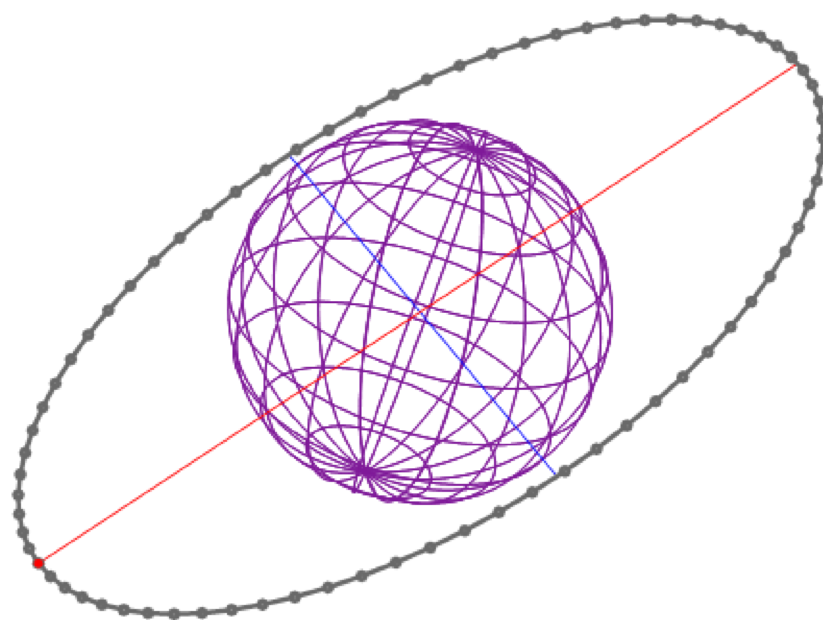
Typický požadavek na editor bývá i možnost zobrazení scény shora, tedy opačným směrem, než ukazuje vektor $\vec{u}$ kamery. Postup je podobný, až na odlišné pořadí konstrukce kvaternionu a tvar vektorů k příslušným osám editoru. U obou konstrukcí rotační matice však musíme ošetřit speciální případy, kdy vektory kamery směřují stejným směrem jako vektory os, s nimiž vytváříme kvaternion.

4.2. Body elipsy

V editoru potřebujeme na dvou různých místech vykreslit dráhu elipsy. V prvním případě při kreslení drátěného modelu koule (což je speciální případ elipsy) a ve druhém u nastavení dráhy animace. V obou případech využíváme parametrické vyjádření elipsoidu. Jeden možný zápis elipsoidu je:

$$\begin{aligned} x &= a \cdot \cos\theta \cdot \cos\phi, \\ y &= b \cdot \cos\phi \cdot \sin\theta, \\ z &= c \cdot \sin\phi, \end{aligned}$$

kde $a, b, c \in \mathbb{R}_0^+$ jsou délky poloos x, y, z. Úhly ϕ a θ jsou sférické souřadnice, kde $\phi \in \left[-\frac{\pi}{2}, \frac{\pi}{2}\right]$ a $\theta \in [-\pi, \pi]$. Většinou však stačí spočítat body pouze u elipsy a transformační maticí ji pak libovolně transformovat. Příklad kreslení elips v editoru RayTracer lze vidět na obrázku 22.



Obrázek 22.: **Vykreslení elipsy v editoru.** Drátěný model koule a dráha animace podél elipsy. Body na elipse značí pozici kamery ve snímcích animace. Směr pohledu kamery se pro každý snímek počítá jako směr od pozice kamery do středu elipsy.

Závěr

Cílem práce bylo představit různé přístupy k optimalizaci ray-tracingu a provést jejich porovnání. Prozkoumána byla široká oblast optimalizací od nejnižší úrovně ray-tracingu, přes optimalizace výpočtů průsečíků paprsku s objekty, optimalizace vržených stínů a techniky akcelerujících struktur. Každá metoda přinesla značné zlepšené oproti naivnímu přístupu ray-tracingu s lineární časovou složitostí pro každý paprsek. Záměrem je snížit složitost na logaritmickou v průměrném případě. Hlavním kritériem při porovnávání metod byl počet všech výpočtů průsečíků objektů s paprskem. Předpoklad naznačoval, že nejvhodnější metoda by měla být technika dělící prostor – kd-tree – vzhledem k její schopnosti přizpůsobení se na rozložení objektů scény. Na námi zkoumaných datech se tato metoda potvrdila také jako nejlepší.

Conclusions

The aim was to present different approaches to optimize ray-tracing and make their comparisons. Was explored broad range of optimization methods from the lowest level ray-tracing through optimization of calculations ray intersections with objects, drop shadows, and optimization techniques accelerating structures. Each method has a significant improvement over the naive ray-tracing approach with linear time complexity for each ray. The intention is to reduce the complexity to logarithmic in the average case. Primary criterion for the comparison of methods is the number of all calculations of the ray intersection with the objects. The assumption indicated that the most appropriate method should be a technique of dividing space – kd-tree – due to its ability to adapt to the distribution of objects in the scene. Even in our survey data, this method was also confirmed as the best.

Reference

[1] AKENINE-MÖLLER, T., HAINES, E., HOFFMAN, N. *Real-Time Rendering*. 3rd ed. 2008. ISBN 978-1-56881-424-7.

[2] BAHOUNEK, O. *Fotorealistické zobrazování*, Bakalářská práce, Univerzita Palackého v Olomouci, 2011.

[3] DUNN, F., PARBERRY, I. *3D Math Primer for graphics and Game Development*. Wordware Publishing, 2002. ISBN 1-55622-911-9.

[4] EBERLY, D. H. *3D Game Engine Design: A Practical Approach to Real-Time Computer Graphics*. Morgan Kaumann Publishers, Academic Press, 2000. ISBN 1558605932.

[5] FERGUSON, R. S. *Practical Algorithms for 3D Computer Graphics*, A. K. Peters Ltd., 2001, ISBN 1-56881-154-3.

[6] HAINES, E. A Proposal for Standard Graphics Environments. *IEEE Computer Graphics and Applications* [online]. 1987, vol. 7, no. 11 [cited 2013-04-04], p. 3–5. Available from: http://www.acm.org/tog/resources/spd/spd.pdf.

[7] KAY, T. L., KAJIYA, J. T. Ray Tracing Complex Scenes. *ACM SIGGRAPH Computer Graphics*, 1986, vol. 20, no. 4, p. 269–278.

[8] KNUTH, D. E. Structured Programming with Goto Statements. *Computing Surveys*, 1974, vol. 6, no. 4, p. 261–301.

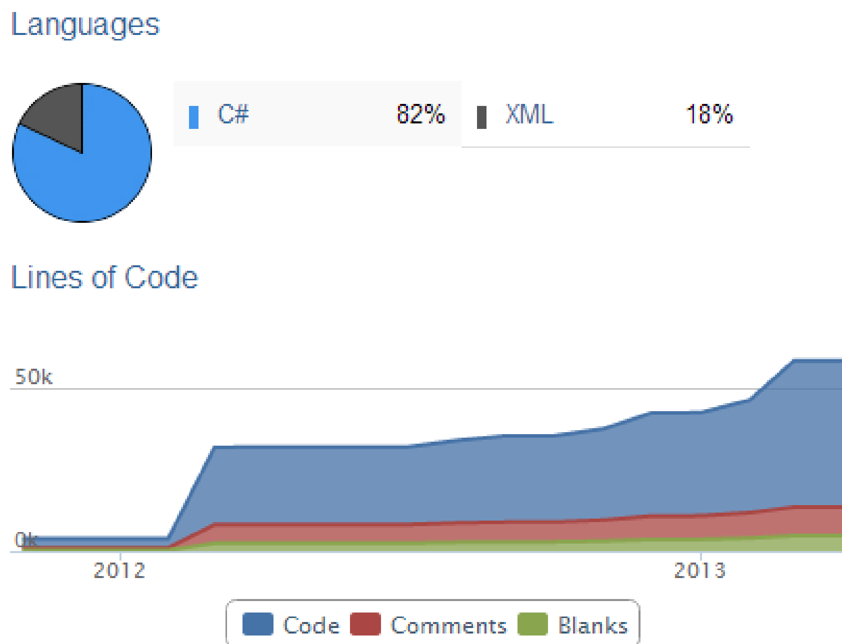
[9] MANOLOPOULOS, Y., NANOPOULOS, A., PAPADOPOULOS, A. N., et al. *R-Trees: Theory and Applications*. 1st ed. London: Springer-Verlag, 2006. ISBN 1-85233-977-2.

[10] PHARR, M., HUMPHREYS, G. *Physically Based Rendering: From Theory to Implementation*. 2nd ed. Elsevier, 2010. ISBN 978-0-12-375079-2.

[11] SLABAUGH, G. G. *Computing Euler angles from a rotation matrix* [online]. 20 August 1999 [cited 28 March 2013]. Available from: http://www.soi.city.ac.uk/~sbbh653/publications/euler.pdf.

[12] WILLIAMS, A., BARRUS, S., MORLEY, R. K., et al. An Efficient and Robust Ray-Box Intersection Algorithm. *Journal of Graphics Tools*, 2005, vol. 10, no. 1, p. 49–54.

[13] ŽÁRA, J., BENEŠ, B., SOCHOR, J., et al. *Moderní počítačová grafika*. 1st ed. Brno: Computer Press, 2004. ISBN 80-251-0454-0.



Obrázek 23.: **Statistika a struktura kódu.**

A. Program RayTracer

Při vývoji programu byla snaha vytvořit uživatelsky příjemný editor jako prostředek k tvorbě složitějších scén ray-tracingu. Program s názvem RayTracer verze 2 je rozšířením programu z práce [2]. Byl napsán v jazyce C# pod platformou .NET 3.5 ve vývojovém prostředí MS Visual Studio 2012. Na počátku druhé verze byly pro zvýšení efektivity vývoje založeny na vývojářských serverech stránky `https://code.google.com/p/raytracer2/` a `https://www.ohloh.net/p/raytracer2`. Postup vývoje programu lze pro zajímavost vidět na obrázku 23.

Struktura programu je rozdělena do tří hlavních částí:

- *Mathematics* poskytuje potřebnou matematiku celému programu,
- *RayTracerLib* je knihovnou pro ray-tracing,
- *EditorLib* je knihovna k trojrozměrnému editoru.

A.1. Ovládání editoru

Ovládání editoru by mělo být intuitivní, používá se výhradně myš. Seznam ovládacích kombinací viz tabulka 7.

ovládání myši	funkcionalita
kolečko myši nahoru/dolů	přiblížení/oddálení v editoru
levé tlačítko myši nad objektem	vybrání objektu nebo zobrazení nabídky objektů
levý dvojklik v editoru	zobrazení aktivního obrázku
podržení pravého tlačítka myši mimo objekt + pohyb myši	přesun středu editoru
podržení pravého tlačítka myši nad objektem + pohyb myši	přesun vybraného objektu
podržení pravého tlačítka myši nad vrcholem obecného objektu + pohyb myši	přesun vybraného vrcholu

Tabulka 7.: **Ovládání editoru.**

A.2. Import a export dat

Každou vytvořenou scénu programu lze exportovat do vnějšího XML souboru. Způsoby načítání scény jsou dva. První kompletně nahradí aktuální scénu externí scénou. Druhý způsob umožňuje přidání externí scény do té současné, kdy po načtení externí scény se zobrazí nabídka objektů, které mají být přidány.

Jednotlivé výstupy programu je možno také ukládat. Obrázky lze z jejich okna po vygenerování uložit ve formátech s příponou PNG, JPEG, BMP nebo i s vlastní příponou. Animace lze ukládat pouze ve formátu AVI nastavením cesty před spuštěním animace v jejím okně vlastností.

A.3. Animace

Animace generuje postupně obrázky a přidává je do videa. K tomuto účelu bylo nutné importovat do programu volně šiřitelnou knihovnu *Splicer.dll* využívající knihovnu *DirectShowLib-2005.dll.* Obě tyto knihovny musí být přítomny v hlavním adresáři programu. Avšak je zde jedno malé omezení pro rozlišení animace. Knihovna umožňuje vytvořit video pouze v určitých rozměrech. Ověřené velikosti jsou pevně přednastavené v editoru.

Animace je v editoru zobrazena tak, jak je vidět na obrázku 22. Body na elipse označují pozice kamery v jednotlivých snímcích. Červený bod je počáteční pozice kamery v animaci.

B. Obsah přiloženého DVD

V samotném závěru práce je uveden stručný popis obsahu přiloženého DVD, tj. závazné adresářové struktury, důležitých souborů apod.

`bin/`
: Instalátor programu RayTracer a další programy spustitelné přímo z CD/DVD. Adresář obsahuje i všechny potřebné knihovny a další soubory pro bezproblémové spuštění programu.

`doc/`
: Dokumentace práce ve formátu PDF, vytvořená dle závazného stylu KI PřF pro diplomové práce, včetně všech příloh, a všechny soubory nutné pro bezproblémové vygenerování PDF souboru dokumentace (v ZIP archivu), tj. zdrojový text dokumentace, vložené obrázky, apod.

`src/`
: Kompletní zdrojové texty programu RayTracer se všemi potřebnými (převzatými) zdrojovými texty, knihovnami a dalšími soubory pro bezproblémové vytvoření spustitelných verzí programu.

`readme.txt`
: Instrukce pro instalaci a spuštění programu RayTracer, včetně požadavků pro jeho provoz.

Navíc DVD obsahuje:

`data/`
: Ukázková a testovací data použitá v práci a pro potřeby obhajoby práce. Data vytvořená programem RayTracer pro demonstraci výsledků. Obsahuje obrázky i videa.

`install/`
: Instalátory aplikací, knihoven a jiných souborů nutných pro provoz programu.

`literature/`
: Některé položky literatury odkazované z dokumentace práce.

U veškerých odjinud převzatých materiálů obsažených na DVD jejich zahrnutí dovolují podmínky pro jejich šíření nebo přiložený souhlas držitele copyrightu. Pro materiály, u kterých toto není splněno, je uveden jejich zdroj (webová adresa) v textu dokumentace práce nebo v souboru `readme.txt`.